

# 8085 Microprocessor Programming

EB-8085

TEXTBOOK

595-4218-04

 **Core** Technology

 **HEATHKIT**<sup>TM</sup>  
E D U C A T I O N A L   S Y S T E M S

Prepare to succeed.<sup>TM</sup>

# 8085 Microprocessor Programming

EB-8085

TEXTBOOK

595-4218-04

## **8085 MICROPROCESSOR PROGRAMMING, Textbook**

**Copyright © 2001, 1998, 1993, 1989** by Heathkit Company, Inc., Benton Harbor, Michigan 49022. All Rights Reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, electronic or mechanical, including photocopying, recording, storage in a data base or retrieval system, or otherwise, without the prior written permission of the publisher.

## INTRODUCTION

The 8085 Microprocessor Programming course is designed to teach you the fundamentals of microprocessors in general and the 8085 instruction set in particular. The fundamental terms and characteristics of all microprocessors are the same. The actual registers and their relationships to each other and the outside world is all that is different.

The 8085 is a general purpose microprocessor. It uses an eight bit data bus and a sixteen bit address bus for memory access and the same data bus and low eight bits of the address bus for input and output operations. The 8085 has 5 hardware interrupt lines, not including the reset. It also has a serial input line and a serial output line.

This course is organized into units which divide the subject of microprocessors into six areas: microprocessor fundamentals, 8085 arithmetic and logic instructions, 8085 data movement instructions, 8085 jump instructions and condition codes, 8085 stack related instructions, and 8085 I/O and interrupt instructions. Notice that the first unit is about microprocessors in general, and all the remaining units specifically discuss the 8085. The experiments for this course are separate, so that you can perform them again and again, whenever desired, without searching through the text. The software for the experiments is provided on the EB-8085-30 ROM cartridge, which plugs into the EWS-8085 trainer system.

The experiments for this course are designed to be performed on the EWS-8085 trainer system. The EWS-8085 trainer system is composed of an ET-3800 trainer with an ETC-8085 CPU module installed. The complete system contains not only the 8085 microprocessor, but a 21 key hexadecimal keypad, a 40 character LCD display (2 rows of 20 characters), 8K bytes of RAM, 24K bytes of ROM, a programmable timer, a 130 connector breadboard, and 8-bit output port, an 8-bit input port, an RS-232C interface, a D/A converter, and an A/D converter. In this configuration, the trainer will accept the ETC-128 E<sup>2</sup>PROM cartridge, which can store sixteen 1K byte programs.

In the trainer documentation, you will see that the microprocessor is referred to more specifically than just 8085. If you were to open the CPU module (And we really wish you wouldn't) you might find that the MPU has still a different identification. Several manufacturers make 8085 equivalent ICs. All have the same registers, bus lines, and instruction set. Therefore, it is easiest to refer to them all by the generic name, 8085. Further, the 8085 is compatible with the 8080 — except that the 8080 cannot perform the read and set interrupt mask instructions (RIM and SIM). Also, it does not have the TRAP, RST 7.5, RST 6.5, and RST 5.5 interrupts, or any of the lines associated with those instructions and interrupts.

The 19 experiments that complement the text demonstrate the concepts you learned in the unit. The table of contents for the experiments identifies which unit each is related to. This will help you, when you refer to them after you complete the course.



## **COURSE OBJECTIVES**

After you have completed this course you will be able to:

1. Identify and define the words, terms, and expressions associated with microprocessors.
2. Describe the main components of an elementary microcomputer system and identify their functions.
3. Identify the program controlled registers in an 8085 microprocessor.
4. Develop a program flowchart.
5. Write and edit elementary machine code and assembly language programs for the 8085 using the EWS-8085 Trainer System.
6. Given an 8085 assembly language mnemonic and a table of opcodes, determine the corresponding machine code.

## **COURSE OUTLINE**

### **UNIT 1 — MICROCOMPUTER BASICS**

#### **INTRODUCTION**

#### **UNIT OBJECTIVES**

#### **TERMS AND CONVENTIONS**

Stored Program Concept

Computer Words

Word Length

#### **AN ELEMENTARY MICROCOMPUTER**

Memory

Fetch-Execute Sequence

A Sample Program

#### **EXECUTING A PROGRAM**

The Fetch Phase

The Execute Phase

Fetching the Add Instruction

Executing the ADI Instruction

Fetching and Executing the HLT Instruction

#### **UNIT SUMMARY**

#### **EXPERIMENTS**

UNIT 2 — MICROPROCESSOR ARCHITECTURE

INTRODUCTION

UNIT OBJECTIVES

ARCHITECTURE OF THE 8085 MPU

The Registers

INSTRUCTION SET

ARITHMETIC INSTRUCTIONS

Add

Subtract

SPECIAL ARITHMETIC AND LOGIC OPCODES

Logical AND

Logical OR

Logical Exclusive OR

SHIFT AND OTHER LOGIC OPERATIONS

DAA and CMA

Shifts or Rotates

UNIT SUMMARY

## UNIT 3 — ADDRESSING MODES

### INTRODUCTION

### UNIT OBJECTIVES

### MOVE INSTRUCTIONS

### IMMEDIATE ADDRESSING

#### Assembly Language

#### Immediate Addressing to 16-bit Registers

### STORES AND LOADS

#### Indirect Loads and Stores

#### Direct Loads and Stores

### OTHER REGISTER TRANSFERS

### UNIT SUMMARY

### EXPERIMENTS

UNIT 4 — INTRODUCTION TO PROGRAMMING

INTRODUCTION

UNIT OBJECTIVES

PROGRAMMING LANGUAGES

PLANNING YOUR PROGRAM

Flow Charts

Constructing a Flowchart

Coding

CONDITIONAL AND UNCONDITIONAL JUMPING

Condition Codes, or Flags

Jumps

UNIT SUMMARY

EXPERIMENTS



## UNIT 5 — STACK OPERATIONS AND SUBROUTINES

### CONTENTS

### INTRODUCTION

### UNIT OBJECTIVES

### WHAT IS A STACK

#### The 8085 Stack

### AUTOMATIC STACK ACTIVITY AND SUBROUTINES

### INSTRUCTIONS THAT CHANGE THE STACK

### UNIT SUMMARY

### EXPERIMENTS

UNIT 6—INPUT/OUTPUT OPERATIONS AND INTERRUPTS

INTRODUCTION

UNIT OBJECTIVES

OUTPUT OPERATIONS

INPUT OPERATIONS SERIAL I/O

Input

Output

INTERRUPTS

More Interrupts

Reset or Restart

UNIT SUMMARY

EXPERIMENTS

UNIT EXAMINATION

EXAMINATION ANSWERS

APPENDIX A—THE 8085 INSTRUCTION SET

APPENDIX B—THE 8085 DATA SHEET

APPENDIX C—EXERCISE PROGRAM LISTINGS

INDEX

*Unit 1*

# **MICROCOMPUTER BASICS**

## CONTENTS

INTRODUCTION . . . . .	1-3
UNIT OBJECTIVES . . . . .	1-4
TERMS AND CONVENTIONS . . . . .	1-5
Stored Program Concept . . . . .	1-6
Computer Words . . . . .	1-8
Word Length . . . . .	1-8
AN ELEMENTARY MICROCOMPUTER . . . . .	1-12
Memory . . . . .	1-16
Fetch-Execute Sequence . . . . .	1-20
A Sample Program . . . . .	1-21
EXECUTING A PROGRAM . . . . .	1-26
The Fetch Phase . . . . .	1-27
The Execute Phase . . . . .	1-32
Fetching the Add Instruction . . . . .	1-34
Executing the ADI Instruction. . . . .	1-36
Fetching and Executing the HLT Instruction. . . . .	1-37
UNIT SUMMARY . . . . .	1-39
EXPERIMENTS . . . . .	1-42

## INTRODUCTION

A microprocessor is a very complex electronic circuit responsible for the programmed arithmetic and logic operations of a microcomputer system. It consists of hundreds of thousands of microscopic transistors squeezed onto a tiny chip of silicon that is often no more than one-eighth inch square. The chip is wired into an integrated circuit (IC) package usually containing 40 or more leads.

The thousands of transistors that make up the microprocessor are arranged to form many different kinds of circuits within the chip. From the standpoint of learning how the microprocessor operates, the most important circuits on the chip are registers, counters, and decoders. In this unit, you will learn how these circuits can work together to perform simple but useful tasks.

In addition to the standard circuits characteristic of all microprocessors, the 8085A that you will be learning to program in this course also contains a serial I/O circuit. This makes it easier to connect to a serial terminal circuit.

As you learn and progress through this unit (and course) you should be aware of several things happening. You will find yourself becoming more familiar with standard microprocessor and programming terms. Many of these apply to all microprocessors in general, but some are very specific to the 8085 as used in the ETC-8085 CPU module when inserted into the ET-3800 trainer. This is referred to as the EWS-8085 Microprocessor Trainer System. You will be learning and using the programming instructions of the 8085A. You should recognize that many other microprocessor units (other than the 8085A) have equivalent instructions to perform the same kind of operations as those you will be learning here. In this sense, the programming proficiency that you gain in this course will enable you to understand and program a wide variety of other microprocessors. You will also find that although there are many different kinds of microprocessors available, many common or similar hardware features exist from one to another. The learning you begin here should be an ongoing process.

If you have previously completed another Heathkit Educational Systems introductory microprocessor course (for the 6800, 6809, or 6811 microprocessors) you may want to continue with Unit 2. Unit 1 of this course presents the same material as those other courses.



## UNIT OBJECTIVES

When you complete this unit you will be able to:

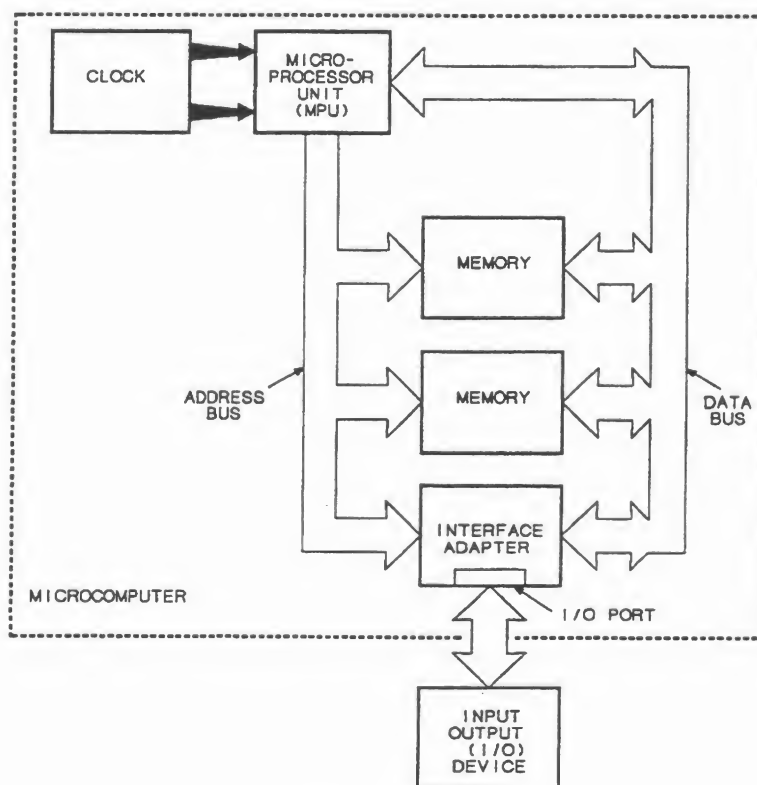
1. Recognize and explain the differences between a microprocessor and microcomputer in reference to their respective block diagrams.
2. Define the terms: microprocessor, microcomputer, input, output, I/O, I/O device, I/O port, instruction, program, stored program concept, word, byte, MPU, ALU, operand, memory, address, read, write, RAM, fetch, execute, MPU cycle, mnemonic, opcode, and bus.
3. Explain the purpose of the following circuits in a typical microprocessor: accumulator, program counter, instruction decoder, controller-sequencer, data register, and address register.
4. Using a simplified block diagram of a hypothetical microprocessor, trace the data flow that takes place between the various circuits during the execution of a simple program.
5. Write simple straight-line programs that can be executed by the ET-3800 Microprocessor Trainer with the ETC-8085 CPU module installed.

## TERMS AND CONVENTIONS

A **microprocessor** is a logic device that is used in digital electronic systems. It is also being used by hobbyists, experimenters and low-budget research groups as a low-cost, general purpose computer. But a distinction should be made between the microprocessor and the microcomputer.

The microprocessor unit, or **MPU**, is a complex logic element that performs arithmetic, logic, and control operations. In most cases it is a single integrated circuit.

A **microcomputer** contains a microprocessor, but it also contains other circuits such as memory devices to store information, interface adapters to connect it with the outside world, and a clock to act as a master timer for the system. Figure 1-1 shows a typical microcomputer. The arrows represent conductors over which binary information flows. The wide arrows represent several conductors connected in parallel. (Parallel circuits have a separate path for each signal.) A group of parallel conductors that carry information is called a **bus**.



**Figure 1-1**  
A Basic Microcomputer.

The microcomputer is composed of everything inside the dotted line. Everything outside the dotted line in Figure 1-1 is referred to as the **outside world**, and all microcomputers must have some means of communicating with it. Information received by the microcomputer from the outside world is referred to as **input** data. Information transmitted to the outside world from the microcomputer is referred to as **output** data. Input information may come from devices like disk drives, various kinds of transducers, mechanical switches, keyboards, or even other computers. Output information may be sent to video displays, output printers, disk drives, or indicator lamps. Some devices such as modems can serve as both an input and an output device. These devices are referred to as **input/output** or **I/O** devices. The point at which the I/O device connects to the microcomputer is called an **I/O port**. The construction and use of I/O ports, though beyond the scope of this course, is central to the subject of microprocessor interfacing and applications.

### Stored Program Concept

A microcomputer is capable of performing many different operations. It can add and subtract numbers and it can perform logical operations. It can read information from an input device and transmit information to an output device. In fact, depending on the microprocessor used, there may be several hundred different operations that the microcomputer can perform.

In spite of all these capabilities, it will do nothing of its own accord. It will only do what it has been told to do, nothing more and nothing less. You must tell the computer exactly what operations to perform and the order in which it should perform them. The operations the computer can be told to perform are called **instructions**. A few of the most common instructions are ADD, SUBTRACT, LOAD REGISTER, STORE REGISTER, MOVE DATA, and JUMP (to another sequence of instructions).

A group of instructions that cause the computer to perform a specific task is called a **program**. One who writes these instructions is called a **programmer**. To design equipment based on a microprocessor, the engineer must know how to program that microprocessor. To repair microprocessor based equipment, a technician must understand exactly what the program is doing.

Programs can be short or long depending on the complexity of the task to be done. A program to add a sequence of numbers might have only a few dozen instructions, but a program to control all the traffic lights in a city would have over a thousand.

A computer is often compared to a calculator, which is controlled by the keyboard. Even inexpensive calculators can perform several operations that can be compared to instructions in a computer. By depressing the right keys, you can instruct the calculator to add, subtract, multiply, divide, and clear the display. Of course, you must also enter the numbers that are to be added, subtracted, etc. With a calculator, you can add a list of numbers as quickly as you can enter the numbers and the instructions. That is, the operation is limited by the speed and accuracy of the operator.

From the start, computer designers recognized that it was the human operator that slowed the computation process. To overcome this, the **stored program concept** was developed. Using this approach, the program is stored in the computer's memory. Suppose, for example, that you have 20 numbers that are to be manipulated by a program that is composed of 100 instructions. Let's further suppose that 10 answers will be produced in the process.

Before any computation begins, the 100-instruction program plus the 20 numbers are loaded into the computer's memory. Furthermore, 10 memory locations are reserved for the 10 answers. Only then is the computer allowed to execute the program. The actual computation time might be less than one millisecond. Compare this to the time it would take to manually enter the instructions and numbers, one at a time, while the computer is running. This automatic operation is one of the features that distinguishes the computer from the simple, non-programmable, calculator. However, the numbers have to be entered in either case. Therefore, stored programs save time only if the operation needs to be repeated or if there are many computations to be performed on the same set of numbers.

## Computer Words

All data is stored in the computer in the form of 1's and 0's. These 1's and 0's are called binary digits, or **bits**. These bits are represented by such physical things as magnetic fields and voltages. Because a bit can represent so little, the bits are grouped together. In computer terminology, a **word** is a group of binary digits that can occupy a storage location. Although the word is made up of several binary digits, the computer handles each word as if it were a single unit. Therefore, the **word** is the fundamental unit of information used in the computer.

## Word Length

In the past several years, a wide variety of microprocessors have been developed. Their cost and capabilities vary widely. One of the most important characteristics of any microprocessor is the word length it can handle. This refers to the length in bits of the most fundamental unit of information.

Today, there are many 16-bit microprocessors, and the most common word length is 16 bits. As a result, the term **word** has come to mean 16 bits. In spite of this, and because 8-bits is both a useful size and an historically common size, many operations are based on 8 bits, which is called a **byte**. Various computers are identified by the number of bits they can work with. As a result, some are called 8-bit machines and others are called 16-bit machines.

Because computers operate in binary, it is often desirable to give numbers in binary (base 2) or hexadecimal (base 16). To avoid confusion, subscript numbers 2, 10, and 16 are used to indicate the base of the number system used. The lowest 8-bit binary number is 0000 0000<sub>2</sub> or 00<sub>16</sub>. The highest is 1111 1111<sub>2</sub> or FF<sub>16</sub>. In decimal numbers, this is the range from 0 to 255<sub>10</sub>. Therefore, a byte can have any one of 256<sub>10</sub> unique values. Therefore, a byte can specify positive numbers between 0 and 255<sub>10</sub>. Or, if the byte represents an instruction, it can be any one of 256 possible operations. It is also common for a byte to represent a character or printer operation. When the high bit is used as a sign (0 for positive or 1 for negative) the byte can represent numbers from -128 to +127.



Naturally, you must have some list of what all the byte patterns represent. The most commonly accepted list is **ASCII**, the American Standard Code for Information Interchange. However, ASCII provides only for the first  $128_{10}$  values (0 through  $127_{10}$ ). The greater values  $128_{10}$  through  $255_{10}$  do not have standard ASCII values. You will find a HEX-ASCII table on the Assembly Language Reference card.

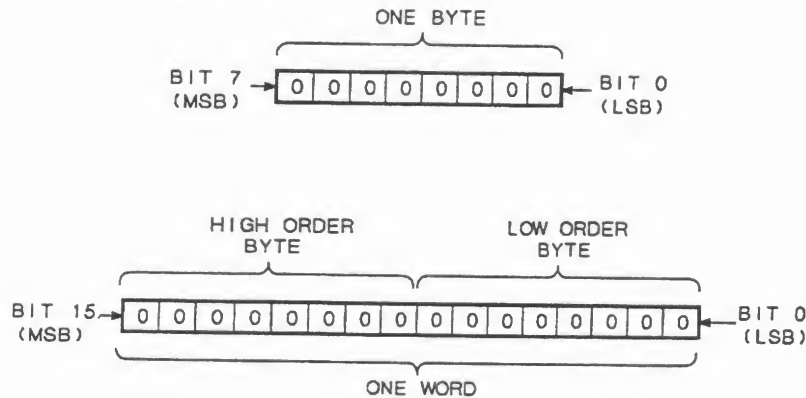
Let's look at an example. In ASCII, the byte  $0100\ 0001_2$  represents the letter "A". On some computers,  $1100\ 0001_2$  also represents an "A," but on others it is used for a graphic character. As you can guess, this is a cause for some confusion when values are transported from one computer to another.

Even within a specific computer, the same byte pattern can have many meanings as mentioned. It can represent a character, a number, or an instruction. You as the programmer, must ensure that an ASCII character or a binary number is not mistaken for an instruction. Later, you will see the consequences of making this mistake.

In a 16-bit machine, ASCII is still commonly used to represent characters. But the 16-bit word allows them to easily work with numbers up to  $65,535_{10}$ . It also allows them to have  $65,535_{10}$  different instructions. However, this also adds to the complexity of the microprocessor. Many microprocessors, such as the 8085, which is the subject of this course, use bytes (8-bits) for data, and 16-bit words for memory addressing. This allows them to access  $65,536_{10}$  memory locations and still keep most operations relatively simple. It is also important to realize that even an 8-bit computer can combine two or more bytes to represent numbers larger than  $255_{10}$ .

The 8-bit value is reflected in the hardware. Within the MPU, there are temporary storage locations called **registers**. The registers are usually byte-length, or multiples of eight bits. Not only are the values within the MPU in groups of eight bits, but the transfer outside the MPU is also in groups of eight. For example, it takes eight wires to transfer a byte of data from the MPU to memory. This group of eight wires is called the **data bus**. The sixteen wires that carry the address are called the **address bus**. Each memory location stores 8 bits.

To manipulate the address values within the MPU, the sixteen bit word is handled in two bytes as shown in Figure 1-2. So that it is easier to discuss the bits in the byte and the bytes in the word, there are some terms you must know. The **least significant bit**, or **LSB**, is the one that has a place value of 1 when the byte represents a number. This is shown as the right most bit in the figure. The **most significant bit** holds the highest position value in the byte or word. This bit has a value of  $128_{10}$  in a byte or  $32768_{10}$  in a 16-bit word. This is also the position that represents the sign (+ or -) in a signed number.



**Figure 1-2**  
Bytes and Words.

To further assist in identifying the bits, each has a number. The LSB is bit 0, because it has a place value of  $2^0$ . The MSB of a byte is bit 7 as shown in Figure 1-2. This is because its place value,  $128_{10}$ , is  $2^7$ . Similarly, in the 16-bit word the MSB is bit 15, because  $32768_{10}$  is  $2^{15}$ . The bytes are also referred to by location. Bits 0 through 7 are the low order byte, or least significant byte. Bits 8 through 15 are the high order byte, or most significant byte.

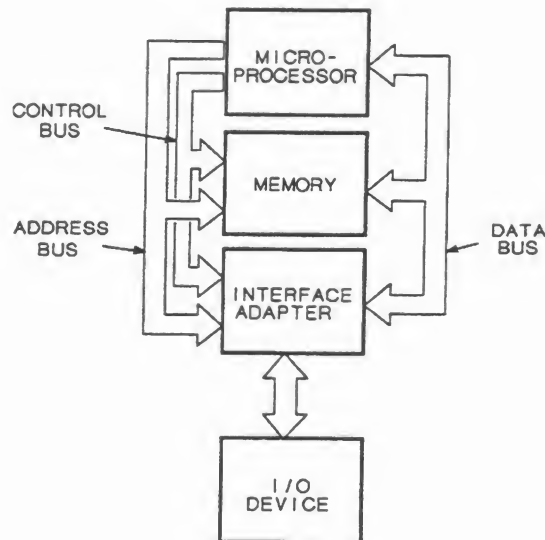
## Self-Test Review

1. Memory and other additional circuits are what distinguish a \_\_\_\_\_ from a microprocessor.
2. A group of parallel conductors that carry information is called a \_\_\_\_\_.
3. Signals received by the microcomputer are called \_\_\_\_\_ signals.
4. The microcomputer sends signals to the "outside world" from its \_\_\_\_\_ ports.
5. A sequence of instructions that perform a specific task is called a \_\_\_\_\_.
6. A binary digit is called a \_\_\_\_\_.
7. The most common computer code for representing the alphabet is called \_\_\_\_\_.
8. Bit 0 is the \_\_\_\_\_ significant bit.
9. A byte consists of \_\_\_\_\_ bits.
10. What is the bit number of the MSB in a byte? \_\_\_\_\_
11. What is the largest number that can be represented by a byte? \_\_\_\_\_
12. What is the largest number that can be represented by a 2-byte word? \_\_\_\_\_
13. The temporary storage locations within the MPU are called \_\_\_\_\_.

## AN ELEMENTARY MICROCOMPUTER

One of the difficulties you may encounter in learning about a microcomputer for the first time is the complexity of its main component -- the microprocessor. The microprocessor may have a dozen or more registers varying in size from 1 to 32 bits. Of course this is not the limit, registers of 128 bits or more could become very common. The microprocessor can have hundreds of instructions, most of which are implemented several different ways. It will have data, address, and control buses. In short, it can be intimidating, if not overwhelming, to start out by considering one of today's full-capability microprocessors.

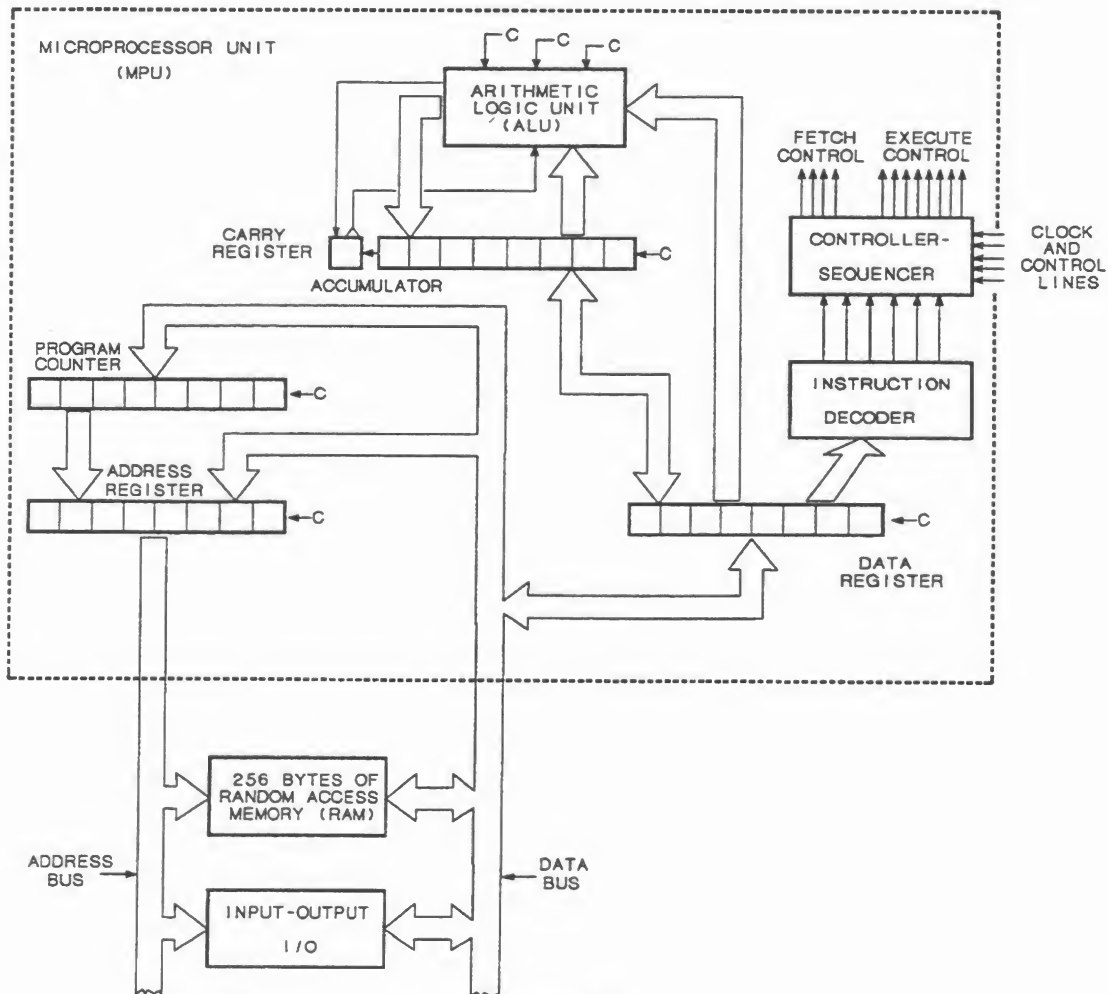
To avoid this problem, we will begin by considering a "stripped-down" version. This will allow you to understand the fundamentals, without getting "bogged-down" in the complexity of the unit. All the characteristics of the more complicated units are embodied in this hypothetical model. To make the transition to the 8085 easier, we have used examples from the 8085's instruction set. The programming examples will actually work on the EWS-8085 Microprocessor Trainer System.



**Figure 1-3**  
The Basic Microcomputer.

A block diagram of a basic microcomputer is shown in Figure 1-3. Its basic elements are the microprocessor, the memory, and the I/O circuitry. For simplicity, we will ignore the I/O circuitry in this unit. In order to do this, we will assume that the program and data are already in memory and that the results of any computations will be held in a register and stored in memory. Ultimately, of course, the program and data must come from the outside world and the results must be sent to the outside world. We will save these procedures until a later unit. This will allow us to concentrate on the microprocessor and the memory.

The microprocessor unit is shown in greater detail in Figure 1-4. For simplicity, only the major registers and circuits are shown. Notice that most of the counters, registers, and buses are 8-bits wide, to accommodate a full byte of data.



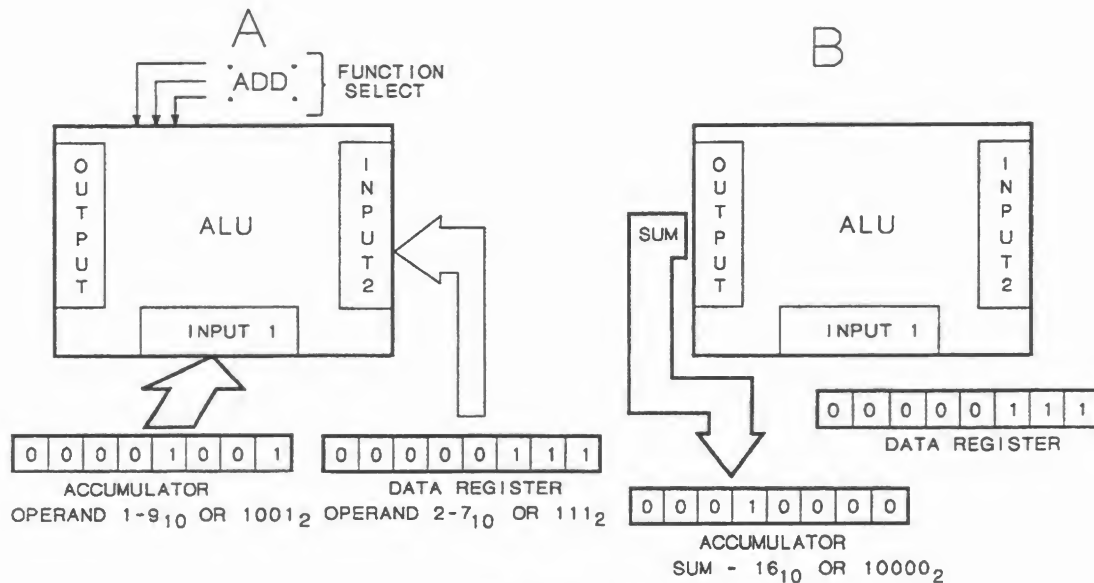
**Figure 1-4**  
A Simplified Microprocessor Unit.



One of the most important circuits is the **arithmetic logic unit (ALU)**. Its purpose is to perform the arithmetic and logic operations on the data that are delivered to it. The ALU has two main inputs. One comes from a register called the **accumulator**, and the other comes from the **data register**. In more complex computers there may be more possible sources, and more than one ALU. The ALU can combine the two pieces of data in only a few ways -- addition, subtraction, or one of the logical operations (such as: OR, AND, and XOR), which will be discussed in Unit 2.

The operation that the ALU performs is determined by signals on the various control lines (Marked C in Figure 1-4) within the MPU.

Generally, the ALU receives one number from the accumulator and another from the data register as shown in Figure 1-5A. Because some operation is performed on these data words, the two inputs are called **operands** or arguments. After the operation is performed, the results are usually returned to the accumulator, which is how that register got its name. For example, assume the two numbers  $7_{10}$  (binary 0000 0111) and  $9_{10}$  (binary 0000 1001) are to be added. Before the numbers can be added, one operand must be placed in the accumulator and the other operand placed in the data register. When the proper ALU control lines are activated, the accumulator and data registers are gated together. A fraction of a second later, the result of that operation (in this example  $16_{10}$ , binary 0001 0000) is routed back into the accumulator, replacing the operand, as shown in Figure 1-5B. Notice that all numbers involved are in binary form.

**Figure 1-5**

The Arithmetic Logic Unit.

The accumulator is the most useful register in the microprocessor. During arithmetic and logic operations it performs a dual function. Before the operation, it holds one of the operands. After the operation it holds the resulting sum, difference, or logical answer. Many operations in the microprocessor involve the accumulator in one way or another.

The data register is a temporary storage location used for many operations. In some microprocessors, the data register is similar to an accumulator, in that it can receive the results of some operations. In our hypothetical microprocessor, the data register also holds the instruction before it is interpreted. You will see how this works a little later in this unit. It is not uncommon for registers to do more than one thing. As indicated by the arrows in Figure 1-4, all data in our hypothetical microprocessor passes through the data register on its way from memory to the accumulator. Data also passes through the data register when transferred from the accumulator back to memory or one of the other registers.

The MPU also contains several other important registers and circuits: the address register, the program counter, the flag register, the index register, the instruction decoder, and the controller-sequencer. Except for the flag and index register, these are shown in Figure 1-4.

The **address register** is a temporary storage location. It holds the address of the memory location or I/O device that is used in the operation presently being performed.

The **program counter** is a register that controls the sequence in which the instructions of the program are executed. Normally, it does this by counting as each instruction is performed, so that it typically contains the address of the next instruction. By changing its contents, you can cause the program to jump, or branch, from one place in memory to another.

The **instruction decoder** is a circuit that interprets the meaning of the byte in the data register when it is a program instruction. The only way it knows that the byte is an instruction is by the timing.

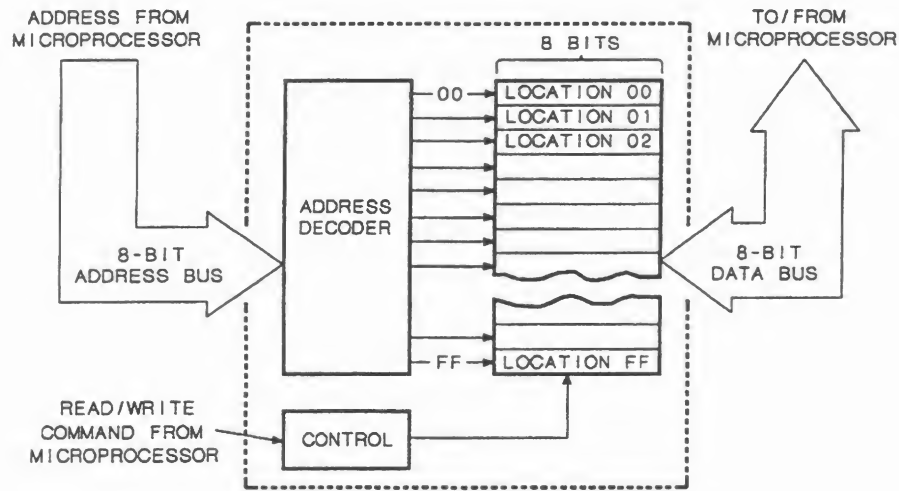
The **controller-sequencer** produces a variety of control signals to carry out the instruction. Because each instruction is different, it has a unique pattern of control signals associated with it. The controller-sequencer produces these patterns after it receives information from the instruction decoder.

Later you will see how these various elements work together to execute simple programs. But first, take a closer look at the memory for our microcomputer.

## Memory

A simplified diagram of the 256-byte read/write memory that is used in our hypothetical microcomputer is shown in Figure 1-6. The memory consists of  $256_{10}$  locations, each of which can store an 8-bit word. This size memory is often referred to as 256 X 8. A read/write memory is one that you can both read data from and write data to with equal ease.

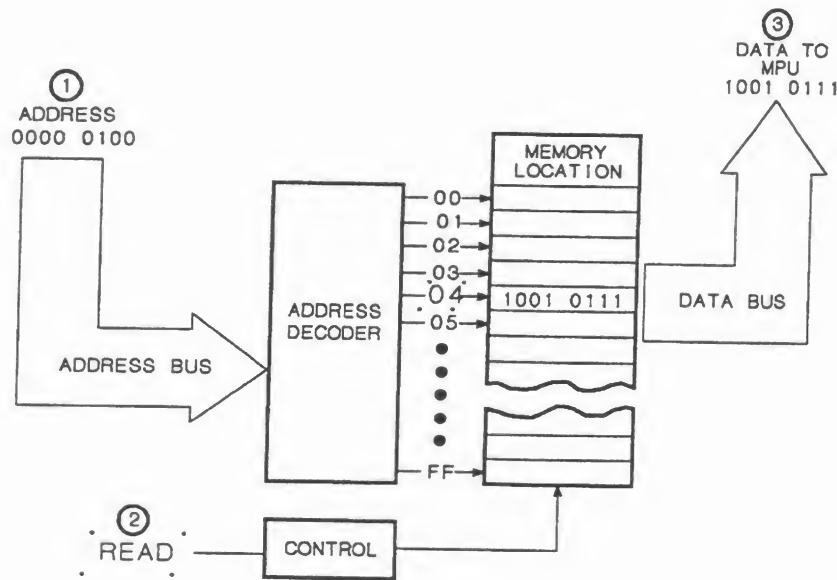
Two buses and a number of control lines connect the memory with the microprocessor unit. The address bus carries the location of the specific byte desired, as specified by the address register, from the MPU to the memory. The control lines indicate whether the byte is to be read or written and when the address lines are valid. The data bus carries the data to the memory during a write operation and from memory to the MPU during a read operation. Notice in Figure 1-6, that the address and control lines are inputs to memory and not outputs. The data bus, on the other hand, has arrows both to and from, because it can be either an input to the data register or an output from the data register.



**Figure 1-6**  
The Random Access Memory.

Each location has a unique identifying number called its address. The first location has address 0. The last in this illustration has address  $255_{10}$ , which is binary 1111 1111, or FF hexadecimal. (Notice that the binary form is written in groups of four bits. This is a common practice that makes it easy to translate the half-bytes, or **nibbles**, to hexadecimal. After you have worked with this notation for a while, it will become very easy for you to read the nibbles as hexadecimal numbers and visualize the hexadecimal in binary.)

In our simplified microprocessor, the value from the address register is always present at the memory, and the read/write signal is normally in the read condition. Therefore, the contents of the selected memory are available on the data bus. However, because of the control leads to the data register, that data is only "read" when it is required by the data register. For a write operation, these same control leads to the data register select when the data register is to output to memory. After both the address and data registers are set to the desired value, the read/write control line selects a WRITE operation, and the data are stored in memory. The read/write line is then returned to the read state and the address and data registers are free to be used for other operations, without affecting the contents of memory.

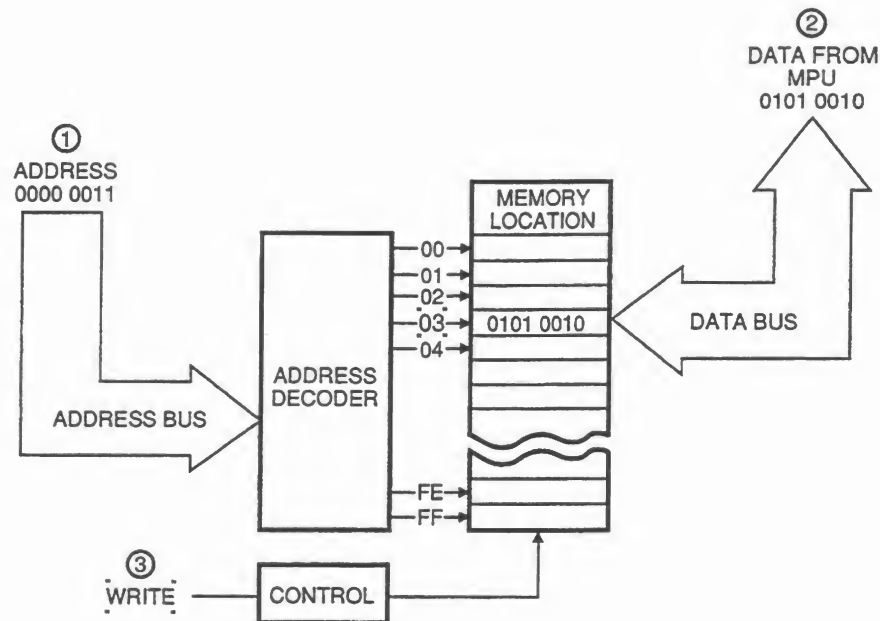


**Figure 1-7**  
Reading from Memory.

Let's look at these procedures a little closer. The read operation is illustrated in Figure 1-7. The MPU places the address of the memory byte desired in the address register. This is applied to the address bus, selecting address  $04_{16}$ . Because the read/write control line is in the READ state, the data from address  $04_{16}$ , in this case  $1001\ 0111_2$  ( $97_{16}$ ), are available on the data bus to the MPU. At this time, the MPU gates the data bus value into the data register, and the memory has been read.

It is important to know that reading memory does not affect the contents of that memory location. This characteristic of not being affected by the read operation is referred to as nondestructive read out (NDRO). It is an important feature, because it allows us to read out the same data as many times as needed. It also saves the time that would be required to write the data back into that memory location, which was necessary in early computers.

More sophisticated microprocessors have additional control lines to select between memory and I/O. Because of the bus structure, the control signal that selects between I/O and memory is applied at the memory and I/O entries to the bus. Therefore, in a read condition, the bus contains which ever input (I/O or memory) is selected.



**Figure 1-8**  
Writing into Memory.

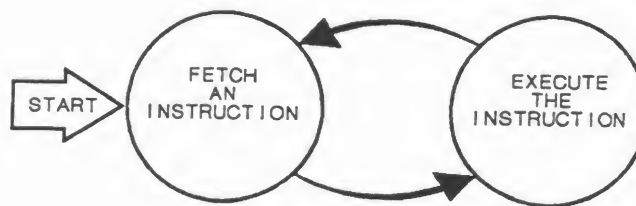
The WRITE operation is shown in Figure 1-8. As in the read operation, the address register applies the desired address on the bus. In this example, the selected address is  $03_{16}$  ( $0000\ 0011_2$ ). The data register applies the correct data  $52_{16}$  ( $0101\ 0010_2$ ) to the data bus. The WRITE signal is applied and the value is gated to the memory cell. Before the data or address buses change, the write signal must be removed. Recall, this returns the bus to a read condition, which is nondestructive, so the memory stays as it was set by the write operation.

This type of memory, in which any address can be read or written to with equal ease is called **Random Access Memory**, or RAM. "Random access" refers to the fact that any address can be accessed at any time. In contrast, some memories, such as "bubble" memory, can only read or write data by sequentially accessing each address location until the desired address is located. Because it is not an efficient way to store data, sequential read memory is not commonly used in microcomputer circuits. The other type of memory most commonly used is called **Read Only Memory**, or ROM. ROM is just as randomly accessible as RAM, but it cannot be easily changed. When you see the term RAM, you must think of **Read And write Memory**. The process for reading ROM is exactly the same as reading RAM. But if you try to write to ROM, the value in memory is not changed.

## Fetch-Execute Sequence

When the microcomputer is executing a program, it goes through a fundamental sequence that is repeated over and over again. Recall that a program consists of instructions that tell the microprocessor exactly what operations to perform. These instructions must be stored in an orderly manner in memory. Typically, this is in the order in which they are to be executed. The instructions are read, or **fetched**, one at a time, from memory by the MPU. After it is fetched, each instruction is interpreted and executed.

The operation of the microprocessor can be broken down into two phases, as shown in Figure 1-9. When the microprocessor is initially started, it enters the **fetch phase**. During the fetch phase, an instruction is taken (read) from memory and decoded by the MPU. Once the instruction is decoded, the MPU switches to the **execute phase**. During this phase, the MPU carries out the operation dictated by the instruction.



**Figure 1-9**  
The Fetch-Execute Sequence.

The fetch phase always consists of the same series of operations. Thus it always takes the same amount of time. However, the execute phase will consist of different sequences of events, depending on what type of instruction is being executed. Thus, the time of the execute phase may vary considerably from one instruction to the next.

## A Sample Program

Now that you have a general idea of the registers and circuits found in a microcomputer, you are ready to examine how all of these circuits work together to execute a simple program. At this point, you are primarily interested in knowing how each step in the process is accomplished. Therefore, the program will be a very trivial one. Longer programs work generally the same way, but with many more instructions.

Let's see how the computer goes about solving a problem like adding seven and ten ( $7 + 10 = ?$ ). While this may seem incredibly easy, the computer has no idea how to solve this problem, until somebody tells it exactly what to do. You must include every detail, because if the right information is not in the right place when it is needed, the result will be wrong.

Before you can write the program, you must know what instructions are available to you and the computer. Every microprocessor has its own instruction set. For this example, assume that after you look over the instruction set you decide that three instructions are necessary to solve this problem. These instructions and a description of what they do are shown in Figure 1-10.

<u>NAME</u>	<u>MNEMONIC</u>	<u>OPCODE</u>	<u>DESCRIPTION</u>
Load Accumulator	MVI A,	0011 1110 <sub>2</sub> or 3E <sub>16</sub>	Load (Move) the contents of the next (Immediate) memory address into the accumulator.
Add	ADI	1100 0110 <sub>2</sub> or C6 <sub>16</sub>	Add the contents of the next (Immediate) memory address to the present contents of the accumulator. The sum will be in the accumulator.
HALT	HLT	0111 0110 <sub>2</sub> or 76 <sub>16</sub>	Stop all operations.

**Figure 1-10**

Instructions used in the simple program.

The first column in the table gives the name of the instruction. When writing programs, it is often inconvenient to write out the entire name. For this reason, each instruction is given an abbreviation or a memory aid called a **mnemonic**. The mnemonics are given in the second column. The third column is called the operation or **opcode**.



This is the binary number that the computer and the programmer use to represent the instruction. The opcode is given in both binary and hexadecimal form. The final column describes exactly what operation is performed when the instruction is executed. Study this table carefully; you will be using these instructions over and over again.

Assume that you wish to add 7 to  $10_{10}$  and place the sum in the accumulator. The program is an elementary one. First, you will load 7 into the accumulator with the MVI A instruction. Next, you will add  $10_{10}$  to the accumulator using the ADI instruction. Finally, you will stop the program with the HLT (halt) instruction.

Using the mnemonics and the decimal representation of the numbers to be added, the program looks like this:

```
MVI      A, 7
ADI      10
HLT
```

Unfortunately, the basic microcomputer cannot understand mnemonics, decimal, or hexadecimal numbers. It can interpret binary numbers and nothing else. Within the computer you use binary. Therefore, these mnemonics and numbers must be converted into binary. You can do this by replacing each mnemonic with its corresponding opcode and each decimal number with its binary counterpart.

That is:

MVI A,7	becomes	0011 1110 opcode from Figure 1-10	0000 0111 binary representation for 7
---------	---------	---	--

and:

ADI 10	becomes	1100 0110 opcode from Figure 1-10	0000 1010 binary representation for $10_{10}$
--------	---------	---	--

Finally,

HLT	becomes	0111 0110 opcode from Figure 1-10
-----	---------	---

Notice that the program consists of three instructions. The first two instructions have two parts: an 8-bit opcode followed by an 8-bit operand. The operands are the two numbers that are to be added (7 and  $10_{10}$ ).

Recall that the microprocessor and memory work with 8-bit words or bytes. Because the first two instructions consist of 16-bits of information, they must be broken into two 8-bit bytes before they can be stored in memory. Thus, when the program is stored in memory, it will look like this:

1st Instruction	1st address	0011 1110	Opcode for MVI A
	2nd address	0000 0111	Operand (7)
2nd Instruction	3rd address	1100 0110	Opcode for ADI
	4th address	0000 1010	Operand (10 <sub>10</sub> )
3rd Instruction	5th address	0111 0110	Opcode for HLT

As you can see, five bytes of memory are required. You can store this 5-byte program any place in memory that is not already being used. Assuming you store it at the first five memory addresses, the memory can be diagrammed as shown in Figure 1-11.

ADDRESS		MEMORY	
HEX	BINARY	BINARY CONTENTS	MNEMONICS/CONTENTS
00	0000 0000	0 0 1 1 1 1 1 0	MVI A
01	0000 0001	0 0 0 0 0 1 1 1	7
02	0000 0010	1 1 0 0 0 1 1 0	ADI
03	0000 0011	0 0 0 0 1 0 1 0	10 <sub>10</sub>
04	0000 0100	0 1 1 1 0 1 1 0	HLT
.			
.			
.			
FD	1111 1101		
FE	1111 1110		
FF	1111 1111		

**Figure 1-11**  
The Program in Memory.

Notice that each memory location has two 8-bit binary numbers associated with it. One is its address, the other is its contents. Be careful not to confuse these two numbers. The address is fixed. It is established when the microcomputer is built. However, the contents may be changed at any time by storing new data.

Before you see how this program is executed, let's review the material covered in this section.

## Self-Test Review

14. The circuit in the microprocessor that performs arithmetic and logic operations is called the \_\_\_\_\_.
15. The numbers that are operated upon by the microprocessor are called \_\_\_\_\_.
16. Before they are added together, the two numbers are in the \_\_\_\_\_ and the \_\_\_\_\_ register.
17. After an arithmetic operation, the result is in the \_\_\_\_\_.
18. The opcode is decoded when the instruction is in the \_\_\_\_\_ register.
19. The memory location to be read or written to is selected by the \_\_\_\_\_ register.
20. The address of the next instruction is normally in the \_\_\_\_\_.
21. An 8-bit address can select any of \_\_\_\_\_ different memory locations.
22. The abbreviation, or memory aid, for each instruction is called a \_\_\_\_\_.
23. The bit pattern that represents the microprocessor instruction is called the \_\_\_\_\_.
24. Memory that can be either read or written during routine computer operation is called \_\_\_\_\_.
25. An instruction is retrieved from memory and decoded during the \_\_\_\_\_ phase.
26. The operation indicated by the instruction is carried out during the \_\_\_\_\_ phase.
27. When the add instruction is executed, the sum will be in the \_\_\_\_\_.

28. How many memory locations are required for the following program? \_\_\_\_\_

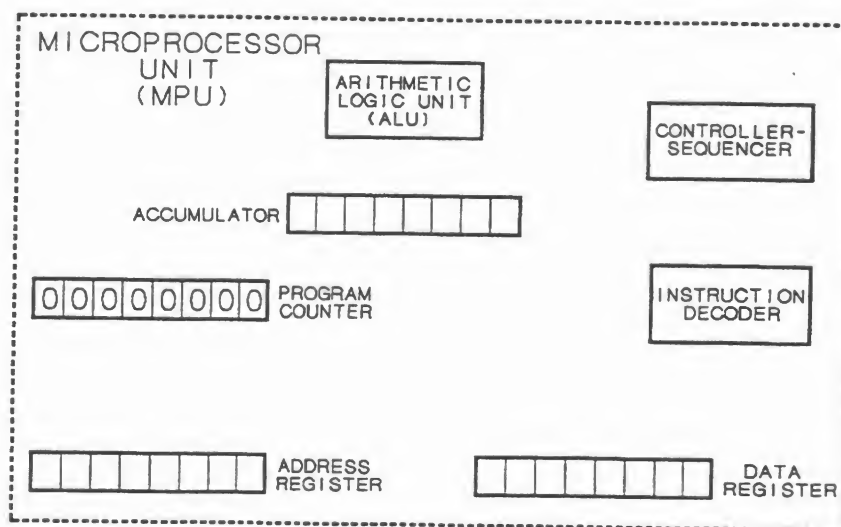
```
MVI      A, 1310
ADI      710
ADI      1010
HLT
```

29. What value will be in the accumulator after this program is executed? \_\_\_\_\_

## EXECUTING A PROGRAM

Before a program can be run, it must be placed in memory. Later, you will see how this is done. For now, assume that the program developed in the previous section is already loaded into memory.

The pertinent registers of the microprocessor are shown in Figure 1-12. Notice that the 5-byte program that adds 7 and  $10_{10}$  is shown in memory addresses zero through four. The following paragraphs and drawings will take you through the step-by-step procedure by which the computer executes this program.



MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS / DECIMAL CONTENTS
0000 0000	0011 1110	MVI A
0000 0001	0000 0111	7
0000 0010	1100 0110	ADI
0000 0011	0000 1010	10
0000 0100	0111 0110	HLT

**Figure 1-12**

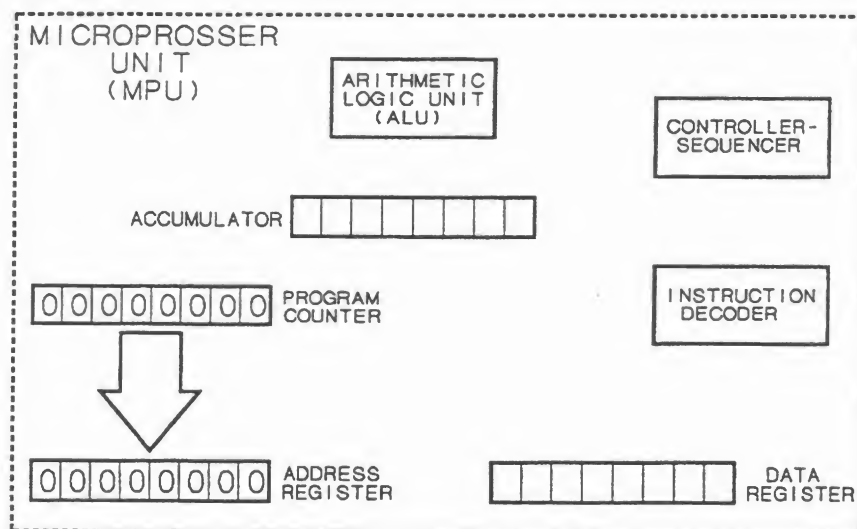
The Program Counter is Set to the Address of the First Instruction.

To begin executing the program, the program counter must be set to the address of the first instruction. In this case, the first instruction is in memory location 0000 0000, so the program counter is set accordingly. The procedure for setting the program counter to the proper address will be discussed later.

## The Fetch Phase

The first step in the execution of any instruction is to fetch the instruction from memory. The sequence of events that happen during the fetch phase is controlled by the controller-sequencer. It produces a number of control signals which will cause the events illustrated in Figures 1-13 through 1-17 to occur.

First, the contents of the program counter are transferred to the address register as shown in Figure 1-13. Recall that this is the address of the first instruction.

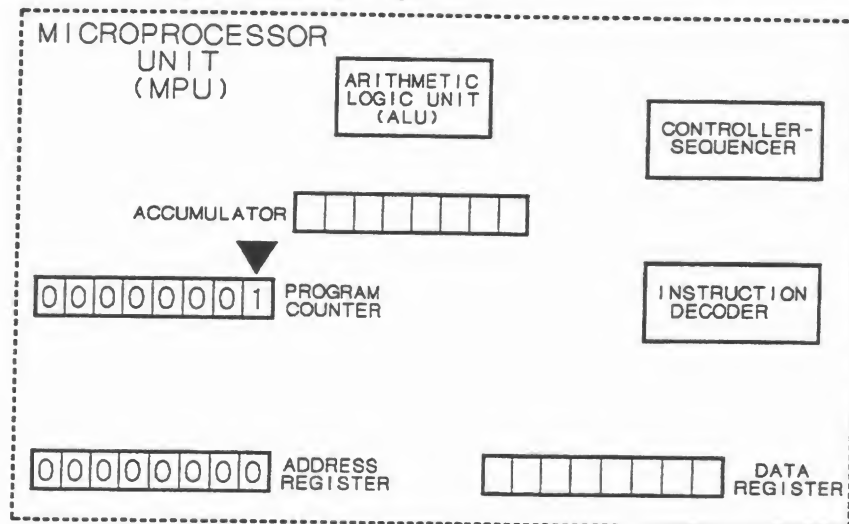


MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/DECIMAL CONTENTS
0000 0000	0011 1110	MVI A
0000 0001	0000 0111	7
0000 0010	1100 0110	ADI
0000 0011	0000 1010	10
0000 0100	0111 0110	HLT

**Figure 1-13**

The Contents of the Program Counter are Transferred to the Address Register.

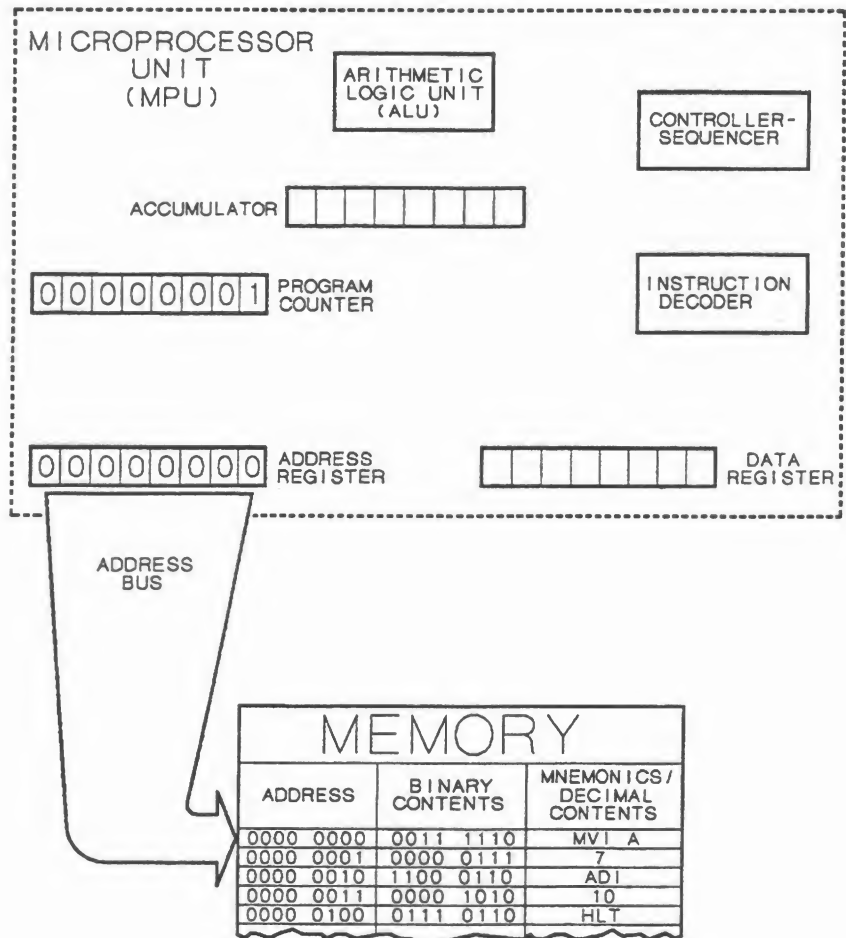
Once the address is safely in the address register, the program counter is incremented by one. That is, its contents change from 0000 0000 to 0000 0001 (Figure 1-14). Notice that this does not change the contents of the address register in any way.



MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS / DECIMAL CONTENTS
0000 0000	0011 1110	MVI A
0000 0001	0000 0111	7
0000 0010	1100 0110	ADI
0000 0011	0000 1010	10
0000 0100	0111 0110	HLT

**Figure 1-14**  
The Program Counter is Incremented.

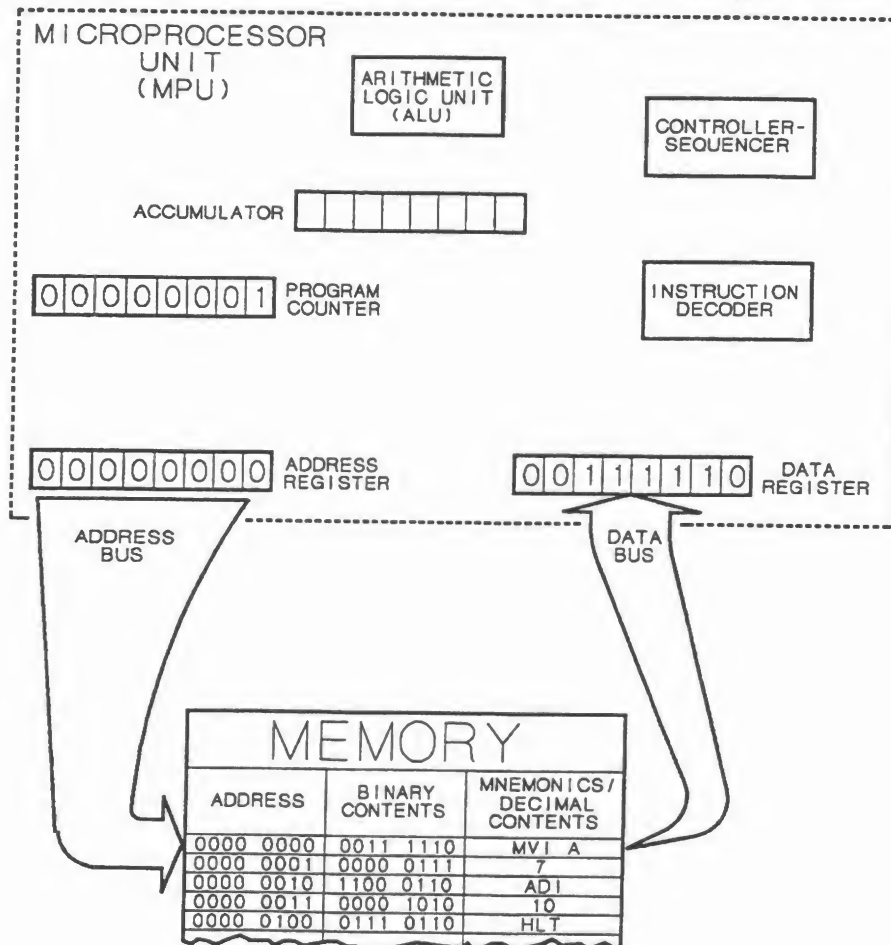
The contents of the address register (0000 0000) are placed on the address bus as shown in Figure 1-15. The memory circuits decode the address to select memory location 0000 0000.



**Figure 1-15**  
The Address of the First instruction is on the Address Bus.



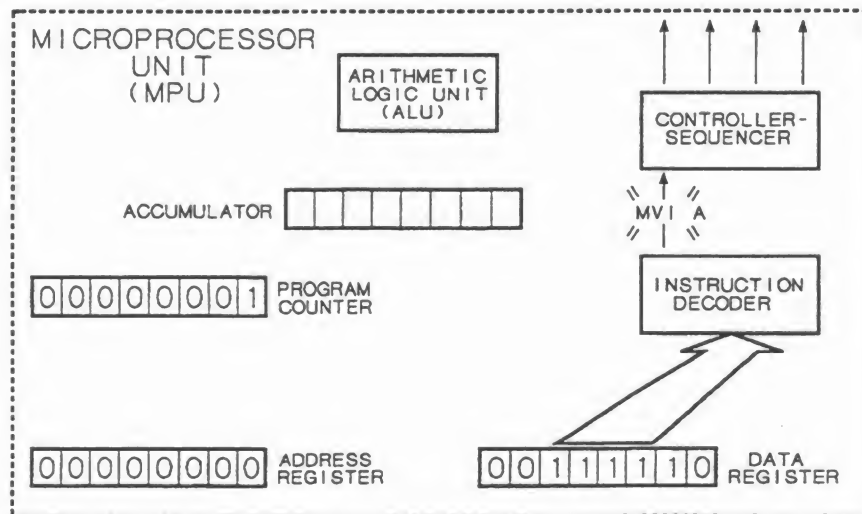
Next, the contents of the selected memory location are read from the data bus into the data register. The data are on the data bus because that address was selected by the address bus. After this operation, the MVI A instruction will be in the data register as shown in Figure 1-16.



**Figure 1-16**

The Opcode for the First instruction is put into the data register.

The next step is to decode the instruction (Figure 1-17). The opcode is transferred to the instruction decoder. This circuit recognizes that the opcode is that of an MVI A instruction. It informs the controller-sequencer of this fact and the controller-sequencer produces the necessary control pulses to carry out the instruction. This completes the fetch phase of the first instruction.



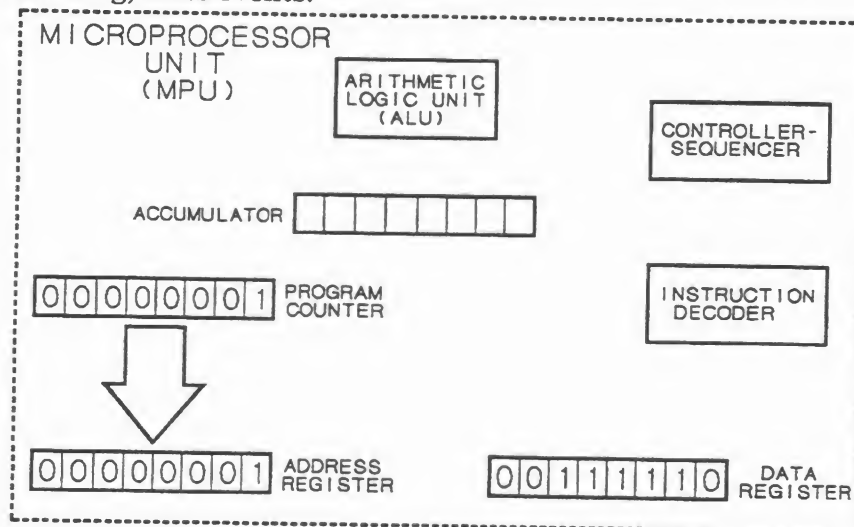
MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS / DECIMAL CONTENTS
0000 0000	0011 1110	MVI A
0000 0001	0000 0111	7
0000 0010	1100 0110	ADI
0000 0011	0000 1010	10
0000 0100	0111 0110	HLT

**Figure 1-17**  
The Opcode is Decoded.

## The Execute Phase

The first instruction was fetched from memory and decoded during the fetch phase. The MPU now "knows" that this is an MVI A instruction. During the execute phase it must carry out this instruction by reading out the next byte of memory and placing it in the accumulator.

The first step is to transfer the address of the next byte from the program counter to the address register (Figure 1-18). You will recall that the program counter was incremented to the proper address (0000 0001) during the previous fetch phase. Notice, too, that the MPU cannot change the address register until the previous byte from memory is latched into the data register. The key to the operation of any MPU is the timing, or coordinating, of its events.

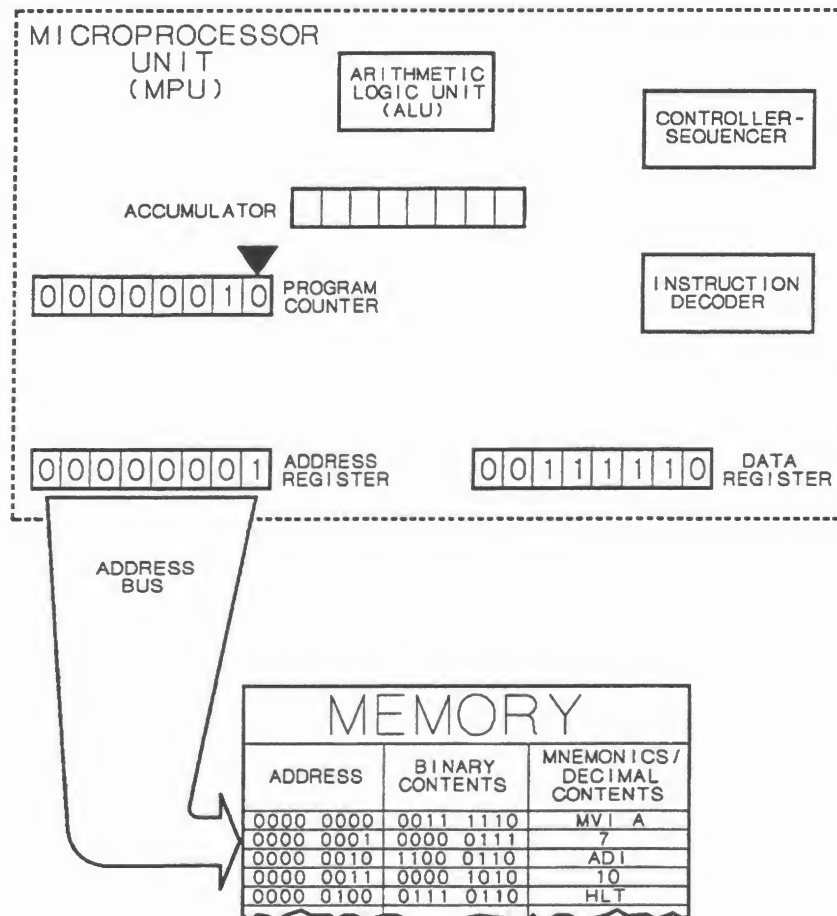


MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/DECIMAL CONTENTS
0000 0000	0011 1110	MVI A
0000 0001	0000 0111	7
0000 0010	1100 0110	ADI
0000 0011	0000 1010	10
0000 0100	0111 0110	HLT

**Figure 1-18**

The Contents of the Program Counter are Transferred to the Address Register.

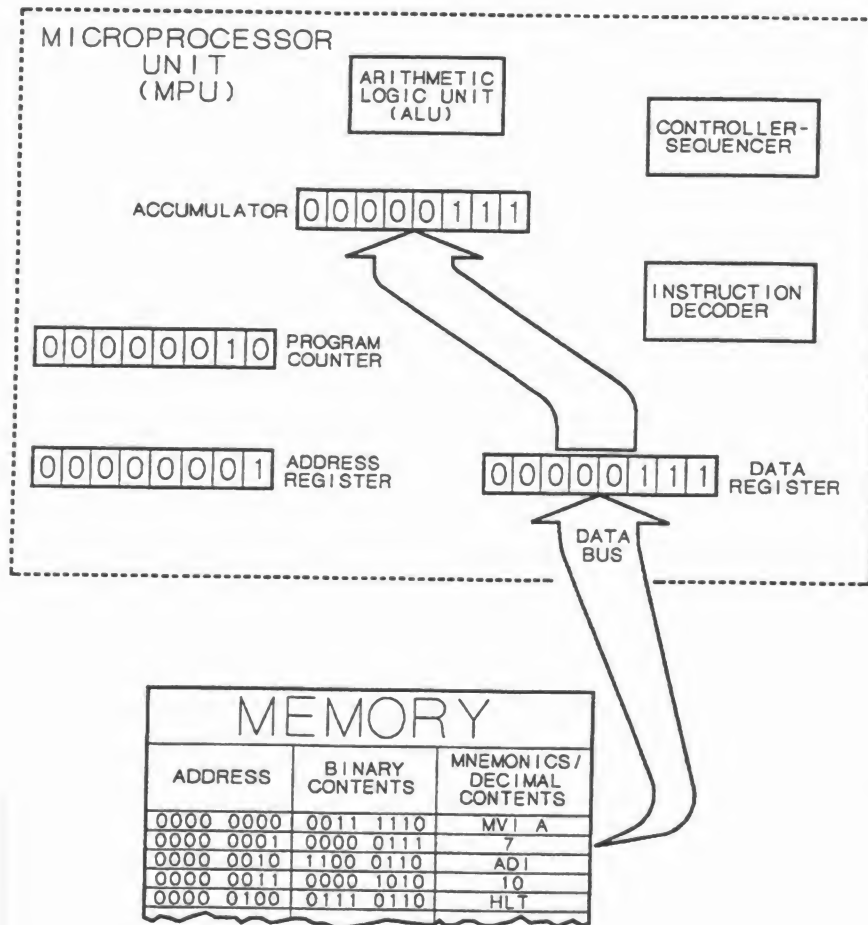
The next operation is shown in Figure 1-19. The program counter is incremented to 0000 0010 in anticipation of the next fetch phase. Notice that the address from the address register is shown on the address bus.



**Figure 1-19**

The Program Counter is incremented and the Contents of the Address Register is on the Address Bus.

The address is decoded and the contents of memory location 0000 0001 are loaded into the data register as shown in Figure 1-20. Recall that this is the number seven. Just as the opcode was immediately available to the instruction decoder, this operand is immediately transferred to the accumulator. Thus, the first execute phase ends with the number 7 in the accumulator.



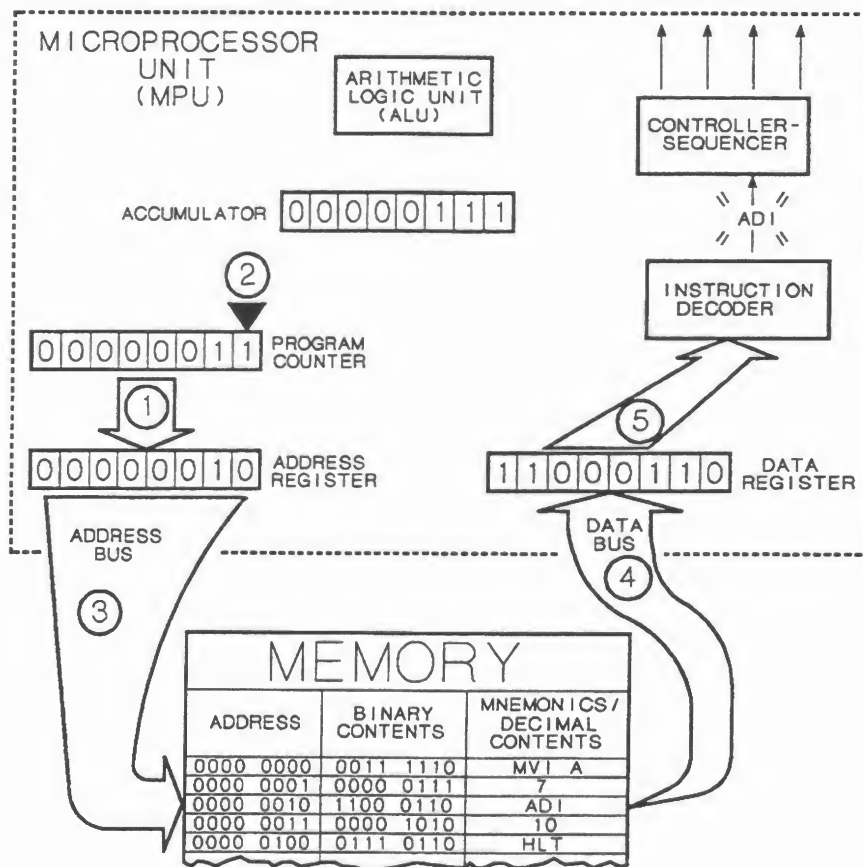
**Figure 1-20**

The First Operand is Transferred to the Accumulator Via the Data Register.

## Fetching the Add Instruction

The next instruction in our program is the ADI instruction. It is fetched from memory using the same procedure that was used for fetching the MVI A instruction. Figure 1-21 illustrates this. The five significant events are as follows:

1. The contents of the program counter (0000 0010) are transferred to the address register.
2. The program counter is incremented to 0000 0011.
3. The address is on the address bus.
4. The contents of the selected memory location are transferred to the data register.
5. The contents of the data register are decoded by the instruction decoder.



**Figure 1-21**  
Fetching the ADI Instruction.

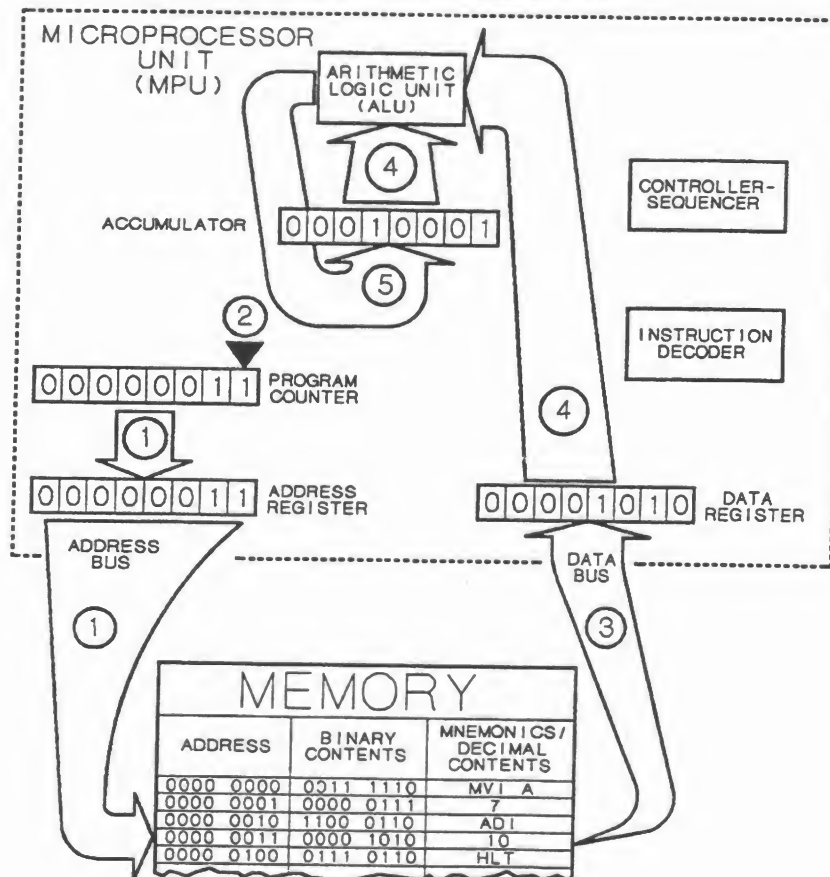
The data word fetched from memory is the opcode for the ADI instruction. Therefore, the controller-sequencer produces the necessary control pulses to execute this instruction.

## Executing the ADI Instruction.

The execution of the ADI instruction is a five step procedure. This procedure is illustrated in Figure 1-22.

1. The contents of the program counter (0000 0011) are transferred to the address register, which places it on the address bus.
2. The program counter is incremented to 0000 0100 in anticipation of the next fetch phase.
3. The operand is read from the data bus into the data register.
4. The ALU combines the values from the accumulator and the data register.
5. The sum from the ALU is placed into the accumulator, replacing the number (7) that was previously stored there.

The computation portion of this program ends with the sum of the two numbers in the accumulator. However, the program is not finished until it tells the computer to stop executing instructions.

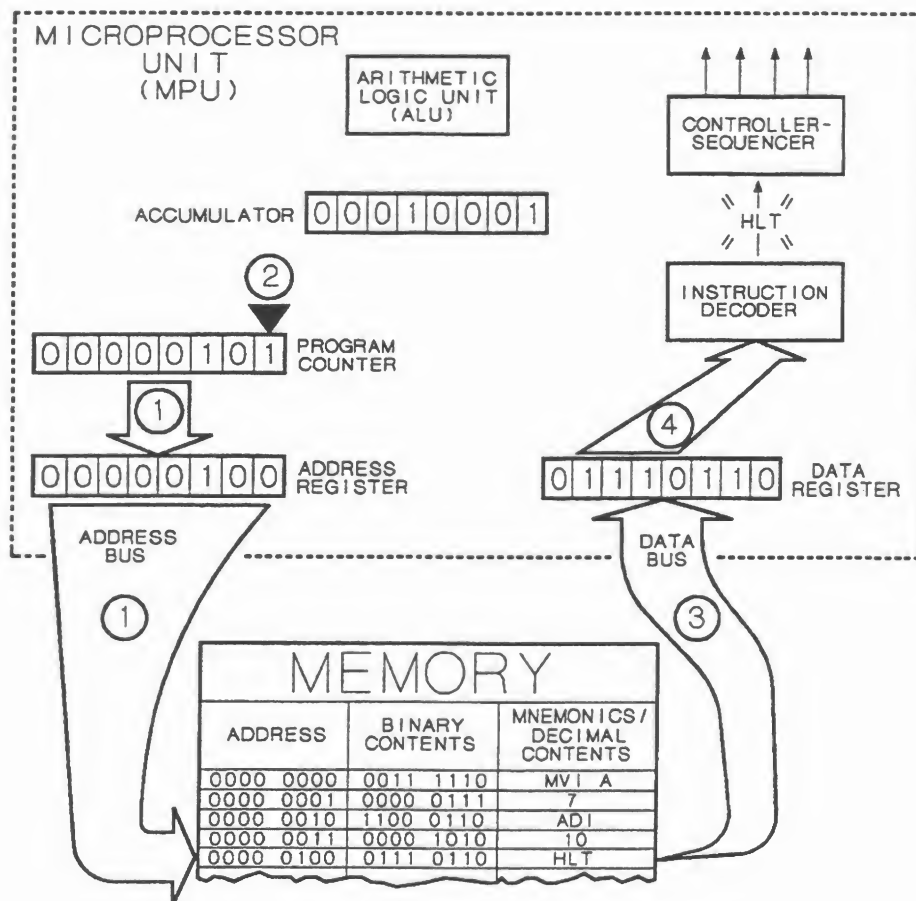


**Figure 1-22**  
Executing the ADI Instruction

## Fetching and Executing the HLT Instruction.

The final instruction in the program is an HLT instruction. It is fetched using the same fetch procedure as before. The four steps are illustrated in Figure 1-23.

1. The address from the program counter is put on the bus via the address register.
- 2.. The program counter is incremented.
3. The contents of memory address 0000 0100 is read into the data register from the data bus.
4. The HLT opcode is translated by the instruction decoder.
5. The MPU halts.



**Figure 1-23**  
Executing the HLT instruction.



In some microprocessors, the execution of the HLT instruction places the computer in a low power consumption or wait mode. The controller-sequencer then stops producing control signals. Consequently, the computer operations stop. Notice that the program has accomplished its objective of adding 7 and  $10_{10}$ . The resulting sum,  $17_{10}$ , is in the accumulator.

## Self-Test Review

Examine this sample program carefully, assume it starts at address  $80_{16}$ , refer to the previous pages, and answer the following questions:

```
MVI      A, 9
ADI      3
ADI      5
HLT
```

30. During the first fetch phase, what binary number is loaded into the data register? \_ \_ \_ \_ \_
31. At the end of the first execute phase, the number 0000 1001 will be in the \_\_\_\_\_ and the \_\_\_\_\_ register.
32. During the second fetch phase, what binary number is loaded into the data register? \_ \_ \_ \_ \_
33. If the first byte of the program is in address  $80_{16}$ , what is the address of the second operand? \_ \_ \_ \_ \_
34. How many bytes does the program occupy? \_\_\_\_\_
35. What number is in the program counter at the end of the second fetch phase? \_ \_ \_ \_ \_
36. When the program is finished running, what number will be in the accumulator? \_ \_ \_ \_ \_
37. What is the final number in the program counter? \_ \_ \_ \_ \_
38. What is the final value of the address register? \_ \_ \_ \_ \_
39. What is the final value in the data register? \_ \_ \_ \_ \_

## UNIT SUMMARY

1. A microprocessor is a logic device that is used in digital electronic systems.
2. The microprocessor is the “brains” of the microcomputer. It performs arithmetic, logic, and control operations.
3. The microcomputer is composed of a microprocessor, RAM, ROM, a clock, and an I/O interface.
4. Parallel conductors within the microcomputer that carry address and data information are called buses.
5. The microcomputer communicates with the “outside world” through one or more I/O ports.
6. The operation of the microprocessor is controlled by a list of instructions called a program.
7. One who writes these instructions is called a programmer.
8. The computer program is stored so that it can be used when it is needed to perform an operation. This is known as the stored program concept.
9. A bit is a binary digit. It can have only a value of 1 or 0.
10. A byte is a group of 8 bits.
11. A byte pattern can represent any one of  $256_{10}$  unique values, including: signed numbers from -128 to +127, unsigned numbers from 0 to 255, or the 128 characters and operations defined by ASCII plus 128 other selected characters.
12. A computer word is the fundamental unit of information used in a the computer. Current usage has given the word a length of sixteen bits.
13. Sixteen bits can represent  $65,536_{10}$  unique values.
14. Microprocessors are known by the number of bits in their registers and on their data buses. For example, a microprocessor with a 16-bit data bus is called a 16-bit microprocessor.
15. Most microprocessors have buses and registers that are 8-bits long or multiples of 8-bits.

16. In a 16-bit word, the byte representing the values from  $2^0$  through  $2^7$  is called the low byte, and the byte representing the values from  $2^8$  through  $2^{15}$  is called the high byte.
17. The bits in a word are numbered from 0 to 15 according to the value they represent in an unsigned number. In other words, the bit representing  $2^0$  is bit 0 and the bit representing  $2^7$  is called bit 7.
18. The microprocessor (MPU) contains many specialized circuits. These include the data register, the instruction decoder, the controller-sequencer, the arithmetic logic unit (ALU), the program counter, the address register, and the accumulator.
- 19. The ALU performs arithmetic or logic operations on the data supplied to the MPU. The data supplied to the ALU are called — operands.
20. The accumulator is a specialized register that performs two functions. First it holds one of the operands prior to an ALU operation. Second, it receives the result of an ALU operation.
21. Specific MPU instructions load data into the various MPU registers, operate on that data, and send data out of the MPU.
- 22. The data register is a temporary storage location for data going to or coming from the data bus.
- 23. The address register is a temporary storage location that controls the address bus. You select a memory location or I/O port by putting its address in the address register.
24. The program counter controls the sequence of instructions in a program. It is incremented after each memory operation, and contains the address of the next memory location to be accessed.
- 25. The instruction decoder translates the opcode in the data register and directs the operation of the controller-sequencer.
26. The controller-sequencer determines the sequence of events necessary to complete the operation described by the instruction decoder.
27. Memory is a series of storage locations outside the MPU. If the MPU is an 8-bit device, each memory location typically stores one byte.

28. Each memory location is identified by a unique address.
29. Memory is connected to the MPU by an address bus, a data bus, and at least one control line (read/write).
30. Reading the contents of memory does not alter the contents of that address. This is known as nondestructive read out (NDRO).
31. Memory that can be both read and written to during normal computer operations is called random access memory, or RAM.
32. Memory that can only be read (not written) is called read only memory, or ROM.
33. When executing a program, the MPU goes through a fundamental sequence of steps or phases called the fetch-execute sequence.
34. In the fetch phase, the MPU reads and decodes the opcode in memory. This time is the same for all types of instructions.
35. During the execute phase, the MPU performs the operation described by the instruction. This time will vary depending on the operation performed.
36. A mnemonic is a memory aid, or abbreviation. Mnemonics are used by programmers to identify the opcodes and operands.
37. The opcode is the binary pattern that determines the operation the MPU will perform. Each opcode has at least one related mnemonic.
38. An operand is the data the instruction is to use. Operands may be in registers or memory when the instruction begins execution. The result of an operation is NOT an operand.

## **EXPERIMENTS**

Perform Experiments 1 and 2.

*Unit 2*

**MICROPROCESSOR  
ARCHITECTURE**

## CONTENTS

INTRODUCTION . . . . .	2-3
UNIT OBJECTIVES . . . . .	2-4
ARCHITECTURE OF THE 8085 MPU . . . . .	2-5
The Registers . . . . .	2-7
INSTRUCTION SET . . . . .	2-10
ARITHMETIC INSTRUCTIONS . . . . .	2-13
Add . . . . .	2-13
Subtract . . . . .	2-20
SPECIAL ARITHMETIC AND LOGIC OPCODES . . . . .	2-24
Logical AND. . . . .	2-24
Logical OR . . . . .	2-26
Logical Exclusive OR . . . . .	2-28
SHIFT AND OTHER LOGIC OPERATIONS . . . . .	2-31
DAA and CMA . . . . .	2-31
Shifts or Rotates . . . . .	2-35
UNIT SUMMARY . . . . .	2-38

## INTRODUCTION

Now that you understand the basic concepts of microprocessors and computer systems, let's take a look at the 8085 used in the EWS-8085 Trainer System. In this unit you will learn what registers the 8085 contains, how they are related to each other, and how to read an instruction set table.

As you learned in Unit 1, the 8085 is a member of the Intel© family of 8080 related microprocessors. This means that its basic architecture is similar to the 8080, the 8088, the 8086, the 80286, the 80386, and the 80486.

For those of you who are familiar with the Motorola© family of microprocessors, such as the 6800, working with the 8085 will require a new way of thinking. For example; the addressing modes are slightly different. Pairs of data bytes are stored in memory in reverse order (low byte in the low address). The interrupt vectors are in low memory instead of high memory.

If you are learning about microprocessors for the first time, don't let this talk discourage you. While it is true that different manufacturers have different ways of doing things, the principles involved are similar, and the specific instruction set for the particular microprocessor you are working with is all that's important.

This course is about the 8085, and you will understand the 8085 when you complete the course. This unit gives you the fundamentals of the 8085. These are important because you cannot understand the units that follow if you don't understand the fundamentals.



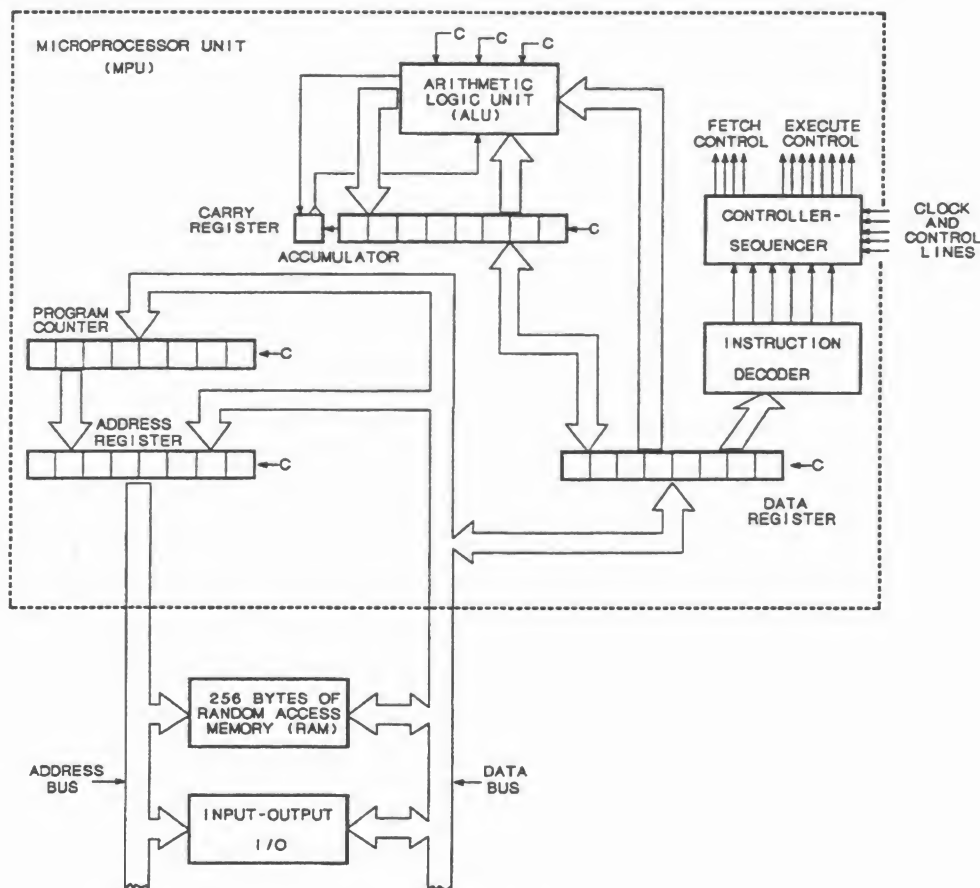
## UNIT OBJECTIVES

When you complete this unit you will be able to:

1. Identify the registers in the 8085, by size, function, and their relationship to each other and the outside world.
2. Draw a block diagram of the 8085 MPU.
3. Given an instruction mnemonic and a table of opcodes, look up the corresponding opcode.
4. Explain the relationship between the accumulator and the flag register.
5. List and identify the flags in the 8085.
6. List the register pairs in the 8085.

## ARCHITECTURE OF THE 8085 MPU

In computer terminology, the word architecture is used to describe the computer's style of construction, its register size and arrangement, its bus configuration, its operating speed, and the access to and by the outside world. The architecture of our hypothetical microprocessor is shown again in Figure 2-1. By the end of this unit, you will be working with block diagrams of the 8085. But, we are using this unit again here because of its simplified architecture, and to make the transition to the real thing.



**Figure 2-1**

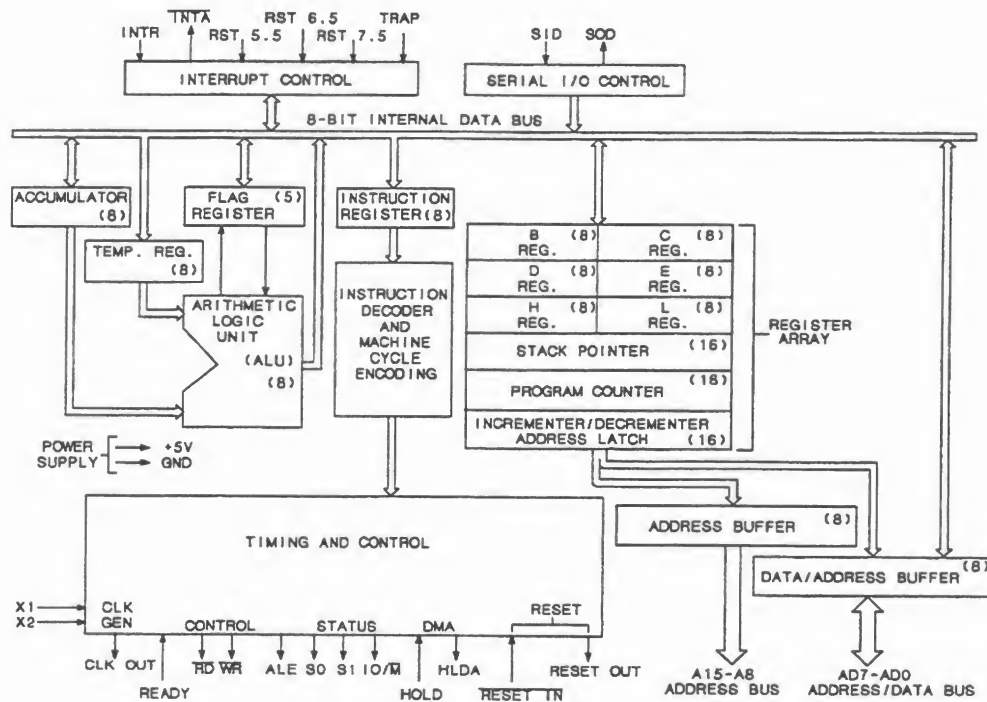
Architecture of the hypothetical microcomputer.

Figure 2-1 illustrates the two important architecture considerations for you as a programmer. First, what registers are there. Second, which registers are linked to which other registers.

You do not need to concern yourself with the control lines, because they only cause things to happen in response to the opcodes, they are not paths for data.

Let's examine the paths in this hypothetical unit. This discussion deals with each register and circuit as seven elements in no particular order. The first element is the accumulator, which can receive input from two sources; the ALU and the data register. The second element, the ALU, receives input from the Data Register, the Accumulator, and the Flags (N, Z, V, and C). What, you may ask, are the Flags? The Flags can be thought of as individual registers that reflect the outcome of the last math or logic operation. You will learn about each one later in this unit. For now, you can think of them as a special register that is linked to the ALU, the accumulator, the data register, and the controller-sequencer - that makes the flags the third element. The fourth element is the program counter. It receives input from the data register, or from memory. Number five, the address register, receives input from the data register, memory, or the program counter. Sixth, the data register gets input from the accumulator, flags, or memory. But, as you can see, the data register can feed any of the other registers. This makes it the most used register, and the second most critical circuit in the microprocessor. The instruction decoder, which only feeds the controller-sequencer, receives its input only from the data register. Finally, the controller sequencer gets its input from the instruction decoder, the clock and external control lines, and the flag register. The controller-sequencer controls the operation of every register in the MPU, which makes it the most critical circuit.

Compare the hypothetical microprocessor (inside the dotted lines in Figure 2-1) to the block diagram of the 8085A shown in Figure 2-2. Notice these elements that are the same: the accumulator, the flags, the ALU, the instruction decoder, the program counter, the address (buffer) register, and the timing and control circuit (which is the same as the controller-sequencer). The data register in our hypothetical MPU is replaced by the dual purpose data/address buffer. In addition, you see an interrupt control circuit, a serial I/O control, a temporary register, an instruction register, a stack pointer, an incrementer/decrementer address latch, as well as registers identified as B, C, D, E, H, and L.

**Figure 2-2**

Block diagram of the 8085 microprocessor.

Beside the name of each register is a number in parentheses, which indicates the number of bits in that register. For example, the accumulator has 8 bits, the flag register has only 5 bits, and the program counter has 16 bits.

During the remainder of this section, you will learn about all of these registers and how they interact. The primary purpose of this course is to teach you the instructions that control data flow through the MPU.

## The Registers

Perhaps the most interesting feature of the 8085 is the group of registers identified as the **register array**. Although all of these registers have special operations associated with them, the letter designated registers B, C, D, E, H, and L are considered general purpose registers. These can be used for certain math operations, as well as serving as pointers to memory addresses. The stack pointer is always a special purpose register that points to a location in memory. You will learn all about the stack operations that use this register in Unit 5.

The 8085 has many instructions that allow you to perform operations on the data in these nine registers. Included are instructions that exchange, copy, and compare their contents. In addition, six of the 8-bit registers are paired to form three 16-bit registers. As implied by the figure, the pairs are BC, DE, and HL. The limited math operations you can perform on these registers make them much less than accumulators, but still more powerful than data registers. The incrementer/decrementer address latch instruction allows you to increase or decrease the value in these registers.

For convenience, the Accumulator is also referred to as the A register. This allows a great amount of similarity in the mnemonics. For example, in Unit 1 you saw how the MVI A instruction placed a value in the accumulator. There are also MVI instructions to place values in the B, C, D, E, H, and L registers.

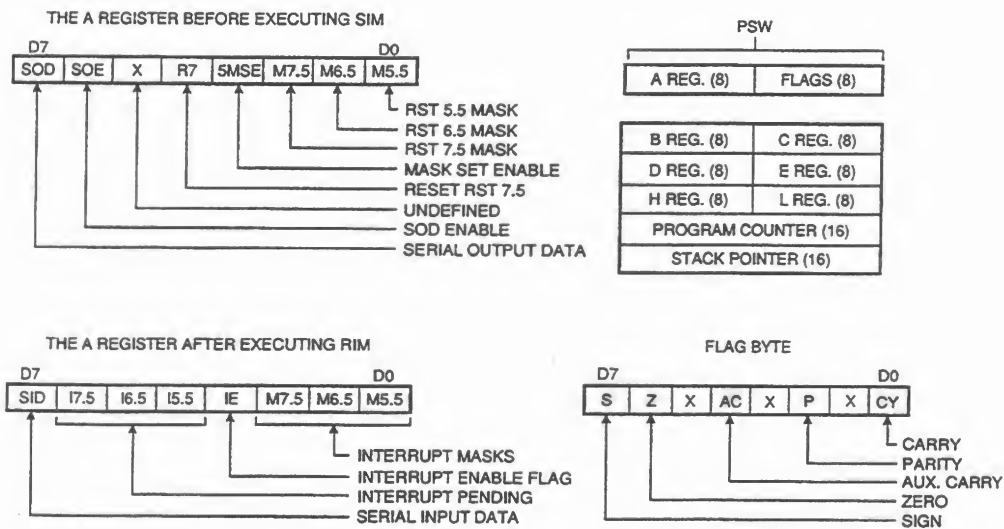
The accumulator and flag register are also paired to form the **processor status word**, or PSW. Special instructions allow you to exchange the PSW with other registers.

As shown by the arrows in Figure 2-2, data flow in the 8085 is quite similar to the hypothetical microprocessor. The address buffer controls the high byte of the address bus, the data/address buffer controls the low byte of the address bus as well as sending and receiving data. This dual purpose architecture requires a circuit outside the MPU to hold the low byte of the address bus so that the data/address buffer can be used to send or receive data. This is also why the MPU has an instruction register and a temporary register to hold one operand for the ALU. Both of these registers are automatic in their operation. As a programmer, you do not need to access either the instruction register or the temporary register.

To make it easier to visualize the registers as you are writing programs, the registers are usually pictured as shown in Figure 2-3. Only the registers that are programmer accessible are shown. Because the accumulator and the flags can be combined to form the PSW, they are shown side-by-side. The bit positions of the flags are identified so that you can easily identify them. In addition, the RIM and SIM values for the accumulator are shown.

You will learn about RIM and SIM later, in the unit on interrupts. For now, however, you should know that the flags are named **sign** (S), **zero** (Z), **auxiliary carry** (A or AC), **parity** (P), and **carry** (C or CY). They will come up periodically during the discussion, and their value will be revealed as you learn about the other instructions. Be careful not to confuse the A flag with the accumulator, which is often called the A register. Similarly, the C flag (carry) should not be confused with the C register.

Locate the similar parts of Figure 2-3 on your programmer's reference card. As you write programs, you will want to refer to this card and these figures regularly. Keep your programmer's reference card handy, you will want to refer to it as you continue this section.



**Figure 2-3**  
Programming model for the 8085.

## INSTRUCTION SET

The key to accessing these registers and the power of the 8085 is the instruction set. Figure 2-4 is a list of the instructions for the 8085. Here, they are listed in a matrix that shows the instruction associated with each of the 256 opcodes. The hexadecimal numbers along the left side represent the high nibble of the opcode. The hexadecimal numbers across the top provide the low nibble. For example, the opcode for RIM is 20<sub>16</sub>. Notice that 10 of the opcodes are not used. This leaves 246 usable opcodes. As the course proceeds, you will see them listed in other ways. At this time, you are not expected to remember all these mnemonics, or even to recognize them. What you should notice, is that they are grouped on the chart.

0	1	2	3	4	5	6	7
0 NOP	LXI B,dbl	STAX B	INX B	INR B	DCR B	MVI B,byte	RLC
1	LXI D,dbl	STAX D	INX D	INR D	DCR D	MVI D,byte	RAL
2 RIM	LXI H,dbl	SHLD adr	INX H	INR H	DCR H	MVI H,byte	DAA
3 SIM	LXI SP,dbl	STA adr	INX SP	INR M	DCR M	MVI M,byte	STC
4 MOV B,B	MOV B,C	MOV B,D	MOV B,E	MOV B,H	MOV B,L	MOV B,M	MOV B,A
5 MOV D,B	MOV D,C	MOV D,D	MOV D,E	MOV D,H	MOV D,L	MOV D,M	MOV D,A
6 MOV H,B	MOV H,C	MOV H,D	MOV H,E	MOV H,H	MOV H,L	MOV H,M	MOV H,A
7 MOV M,B	MOV M,C	MOV M,D	MOV M,E	MOV M,H	MOV M,L	HLT	MOV M,A
8 ADD B	ADD C	ADD D	ADD E	ADD H	ADD L	ADD M	ADD A
9 SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB M	SUB A
A ANA B	ANA C	ANA D	ANA E	ANA H	ANA L	ANA M	ANA A
B ORA B	ORA C	ORA D	ORA E	ORA H	ORA L	ORA M	ORA A
C RNZ	POP B	JNZ adr	JMP adr	CNZ adr	PUSH B	ADI byte	RST 0
D RNC	POP D	JNC adr	OUT port	CNC adr	PUSH D	SUI byte	RST 2
E RPO	POP H	JPO adr	XTHL	CPO adr	PUSH H	ANI byte	RST 4
F RP	POP PSW	JP adr	DI	CP adr	PUSH PSW	ORI byte	RST 6

8	9	A	B	C	D	E	F
0	DAD B	LDAX B	DCX B	INR C	DCR C	MVI C,byte	RRC
1	DAD D	LDAX D	DCX D	INR E	DCR E	MVI E,byte	RAR
2	DAD H	LHLD adr	DCX H	INR L	DCR L	MVI L,byte	CMA
3	DAD SP	LDA adr	DCX SP	INR A	DCR A	MVI A,byte	CMC
4 MOV C,B	MOV C,C	MOV C,D	MOV C,E	MOV C,H	MOV C,L	MOV C,M	MOV C,A
5 MOV E,B	MOV E,C	MOV E,D	MOV E,E	MOV E,H	MOV E,L	MOV E,M	MOV E,A
6 MOV L,B	MOV L,C	MOV L,D	MOV L,E	MOV L,H	MOV L,L	MOV L,M	MOV L,A
7 MOV A,B	MOV A,C	MOV A,D	MOV A,E	MOV A,H	MOV A,L	MOV A,M	MOV A,A
8 ADC B	ADC C	ADC D	ADC E	ADC H	ADC L	ADC M	ADC A
9 SBB B	SBB C	SBB D	SBB E	SBB H	SBB L	SBB M	SBB A
A XRA B	XRA C	XRA D	XRA E	XRA H	XRA L	XRA M	XRA A
B CMP B	CMP C	CMP D	CMP E	CMP H	CMP L	CMP M	CMP A
C RZ	RET	JZ adr		CZ adr	CALL adr	ACI byte	RST 1
D RC		JC adr	IN port	CC adr		SBI byte	RST 3
E RPE	PCHL	JPE adr	XCHG	CPE adr		XRI byte	RST 5
F RM	SPHL	JM adr	EI	CM adr		CPI byte	RST 7

Figure 2-4

8085 Microprocessor Instruction Set.



For example, the block of opcodes from  $40_{16}$  to  $7F_{16}$  are all move (MOV) instructions, except the HLT instruction,  $76_{16}$ , which occupies the spot that might logically have been MOV M,M. It helps to look at the binary values to see how the opcodes are structured. The MOV instructions all have the pattern  $01xx\ xxxx_2$ , where the x values may vary. Further examination shows that bits 3 through 6 designate the letter before the comma and bits 0 through 2 designate the letter after the comma. This is easiest to observe in the low three bits because the moves in columns  $9_{16}$  ( $1001$ ) and  $1_{16}$  ( $0001$ ) all end in the letter C. Not so easily observed is that the same  $001$  pattern in bits 6, 5, and 4 also designates the letter C in opcodes  $48_{16}$  through  $4F_{16}$ .

To see the similarities in the opcodes, you compare the values that are common. In the previous example, the MOV instructions have a first nibble of  $0100$ ,  $0101$ ,  $0110$ , or  $0111$ . In every case the first bit is a 0 and the second bit is a 1, which means the pattern is  $01xx$ . The bits in the second nibble can be any of the sixteen possible patterns, so that has a pattern of  $xxxx$ .

Let's look at some of the other groups of opcodes and see how they are alike, and yet different. Notice the opcodes with the RST mnemonics ( $C7$  to  $F7$  and  $CF$  to  $FF$ ). These all have the form  $11xx\ x111_2$ . The first nibble is C, D, E, or F—all have the first two bits as 1s. The second nibble is either 7 or F, both of which have the last three bits as 1s. Now look at the INR instructions in columns 4 and C. You should see that these all have the following bit pattern,  $0xxx\ x100_2$ . From your knowledge of digital circuits, you should observe that these patterns can be decoded by AND and OR gates to recognize these opcodes when they are in the instruction register. This is significant, because it is exactly how the MPU decides what the opcode is.

On your programmer's reference card the opcodes are divided into four groups: data transfer, arithmetic and logic, branch control, and I/O and Machine control. For this course, we have regrouped the last two categories and diverted two special arithmetic and logic opcodes. As a result, you will learn the opcodes in five categories: arithmetic and logic, data transfer (addressing modes), jumps and condition codes, stack operations, and I/O and interrupts.



## Self-Test Review

1. Refer to Figure 2-4. What is the common pattern for the POP and PUSH opcodes? \_ \_ \_ \_ \_
2. Again looking at the POP and PUSH opcodes, bit 2 is a 1 for the \_\_\_\_\_ opcodes and a 0 for the \_\_\_\_\_ opcodes.
3. The \_\_\_\_\_, or \_ \_ , is a two-byte register formed by a combination of the accumulator and the flag register.
4. In the 8085 the B, C, D, E, H, L, Stack Pointer, Program counter, and increment/decrement address latch form the \_\_\_\_\_.
5. Opcodes with the pattern 10xx x001 all contain the designation for the \_\_\_\_\_ register in their mnemonic.
6. In the 8085, the ALU gets its input from the \_\_\_\_\_, the \_\_\_\_\_ register, and the \_\_\_\_\_.
7. The 8085 instruction decoder decodes the opcode in the \_\_\_\_\_ register.
8. The 8085 has four register pairs, that are designated BC, DE, HL, and PSW.
9. The low byte of the 8085 address bus is controlled by the \_\_\_\_\_ buffer.
10. The high byte of the 8085 address bus is controlled by the \_\_\_\_\_ buffer.

## ARITHMETIC INSTRUCTIONS

The 8085's arithmetic opcodes are discussed in five categories: add, subtract, increment, decrement, and special. Excepting the special instructions, these opcodes do simple addition or subtraction. In addition to discussing the arithmetic operations, you will also learn about the carry flag. The other flags are discussed in Unit 4. In general, the flags are affected by the results of arithmetic and logic operations but not by any other operations. This is very important, because you may want to perform other operations before the conditional operation based on the flags. You must know which instructions affect the flags, and which ones do not. As you read about each instruction, be sure to note which flags are affected. On your programmer's reference card you should notice the asterisk (\*) or \*\*) and crosses (†). There are corresponding notes that indicate which instructions affect all flags, which affect all except the carry flag, and which affect only the carry flag.

### Add

Most of the add instructions increase the value of the accumulator. The increment instructions, which are also in this category have the ability to add one (1) to any specific register or a location in memory. The double add instruction affects the HL register pair. There are seven add instructions. They have the following mnemonics and affects on the flags:

<u>Mnemonic</u>	<u>Flag Impact</u>
ADI	all flags affected
ADD	all flags affected
ACI	all flags affected
ADC	all flags affected
INR	all except CARRY affected
INX	no flags affected
DAD	only CARRY affected

## 2-14 MICROPROCESSOR ARCHITECTURE

---

Before considering the addition of bytes or words, let's review the addition of individual bits. Naturally, 0 plus 0 is 0. 1 plus 0 or 0 plus 1 is 1. If both bits are 1, the result is 0 with a carry of 1 to the next level. Because of the carry generated by two ones, you must also consider the sum of three bits, the two previously considered and the carry from the next lower order. If any of the bits is zero, the addition is just like two bits ignoring the zero. But, if all three bits are 1's, the result is not only the carry to the next level, but a one in that position created by the other two ones. Here is a summary of the rules.

1.  $0+0=0$
2.  $0+1=1$
3.  $1+1=0$  with a carry of 1.
4.  $1+1+1=1$  with a carry of 1.

The ADI and ADD instructions add a byte to the accumulator. This is the addition of two binary numbers, so the only other consideration is if a carry is generated. In the binary addition of two bytes, you cannot have more than a carry of one. Consider the following examples.

carry	11111 111	11111 111	11111 111
	0101 0101	1111 1111	1111 1111
	<u>+1010 1011</u>	<u>+1111 1111</u>	<u>+0000 0001</u>
total	10000 0000	11111 1110	10000 0000

In each of these examples, the combination of bits at each level generate a carry to the next level. Only in the center example did the total contain ones in the intermediate positions. The carry to the eighth position is sent to the carry flag, because there is no room for it in the byte. As a result, each of these examples would leave a 1 in the carry flag. This is fine if you only wish to add two single byte values, but to combine two or more bytes for a result, you typically add the low order bytes and then add the others, working your way to the most significant byte. To accommodate this string of additions, you need to consider the carry from the lower byte when you perform the addition. This is the reason for the ACI and ADC opcodes.

These opcodes add with the carry. In other words, for the 0 bit in that byte, the carry bit is also added. Thus, if there was a carry from the previous arithmetic operation, the low bit in this operation will include that carry. However, you must consider that ADD, ADI, ADC, and ACI all place their result in the accumulator. To use a series of these operations, you would have to store the result of each operation and load a new byte into the accumulator before the next addition. As mentioned earlier, the mathematical operations affect the flags, but the transfer instructions do not. Therefore, your transfers between additions will not change the value of the carry flag from the last addition.

You know that the results of the addition are placed in the accumulator, but where does the other operand come from? Since you remember the block diagram of the 8085 MPU, you probably answered "the temporary register." However, as you may also recall, the temporary register is not shown in Figure 2-3, the programming model. The temporary register is loaded automatically with the operand specified in the add instruction. If the opcode is ADI or ACI, the value to be added is obtained from the byte of memory immediately after the add opcode. For the ADD and ADC instructions, the source for the temporary register value may be any of the 8-bit registers or the contents of memory at address M. However, you cannot add the flag register. Thus, you can specify the ADD or ADC instructions with any of the following:

Mnemonic	Opcode	Action
ADD A	87	$A + A \Rightarrow A$
ADCA	8F	$A + A + cy \Rightarrow A$
ADD B	80	$A + B \Rightarrow A$
ADCB	88	$A + B + cy \Rightarrow A$
ADDC	81	$A + C \Rightarrow A$
ADCC	89	$A + C + cy \Rightarrow A$
ADDD	82	$A + D \Rightarrow A$
ADCD	8A	$A + D + cy \Rightarrow A$
ADDE	83	$A + E \Rightarrow A$
ADCE	8B	$A + E + cy \Rightarrow A$
ADD H	84	$A + H \Rightarrow A$
ADCH	8C	$A + H + cy \Rightarrow A$
ADD L	85	$A + L \Rightarrow A$
ADCL	8D	$A + L + cy \Rightarrow A$
ADDM	86	$A + M \Rightarrow A$
ADCM	8E	$A + M + cy \Rightarrow A$

For convenience, the notations **ADD r** and **ADC r** are used. The **r** indicates you can specify one of the 8-bit registers. Notice that these all have the opcode pattern 1000 xxxx<sub>2</sub> (8x16). Further, bit 3 determines the effect of the carry flag. If bit 3 is a 1, the carry flag is considered. If bit 3 is 0, the carry flag has no effect. Also from the opcodes, it is easy to derive the following chart.

Register	Binary pattern for bits
	<u>210</u>
A	111
B	000
C	001
D	010
E	011
H	100
L	101
M	110

These patterns are common to all instructions that affect the 8-bit registers in the 8085. This is why the columns and rows in the opcode table have the common pattern you saw earlier. Look for these patterns as you learn about the other opcodes.

It is easy to understand the meaning of registers A through L, but what about M? The description above indicates that it means memory. But which memory? This is a very important feature of the 8085 microprocessor. The H (high) and L (low) registers are always pointing to a specific location called M, for memory. Many of the 8085 opcodes deal with M, so it is important to be aware of the value in the HL register pair. Not just HL, but any register pair that points to an address in memory is referred to as an **index**. This is the meaning of the X in INX and DCX.

In the instruction mnemonics, the indexes are specified as B, D, H, and SP. You should recognize B, D, and H as the high byte of those register pairs. SP is the stack pointer register.

With the exception of ADD A, none of these instructions affect the registers specified as the operand of the mnemonic. These are source registers, not destinations. In ADD A, the source is the A register, but as you know, that is also the accumulator. As a result, this instruction adds the accumulator to itself, so obviously the source value (the accumulator) is changed. ADD A is one way to double the value in the accumulator. You will learn the other when you read about the logic operations later in this unit. ADC A not only doubles the accumulator, it also considers the carry (cy) from the next previous arithmetic operation.

ADI and ACI are similar to ADD and ADC, except that the value to be added to the accumulator is in the memory address after the opcode. This is called **immediate** addressing, because the value is immediately after the opcode. Again, the C in ACI indicates that the carry flag will be added to the sum of the accumulator and the immediate value.

INR instructions have the pattern 00xx x100. This time, bits 3, 4, and 5 identify which one of the eight registers is involved. The INR opcode increases the contents of the register named. For example, the INR D instruction increases the D register by 1. You can add two to a register by increasing the register twice. However, you cannot consider the carry flag, because it is not affected by the INR instruction. Therefore, you cannot link together several bytes and add to them with INR instructions.

However, you can increment register pairs: BC, DE, HL, or the stack pointer. To do this you use the INX instruction. Although none of the flags are affected by INX, the register pair is incremented by one. For example; if B is  $0000\ 0000_2$  and C is  $1111\ 1111_2$ , you can use INX B to increase that value by 1. The result will be a B value of  $0000\ 0001_2$  and C value of  $0000\ 0000_2$ . Notice that the carry from the C register was applied to the B register. If both registers contain all 1s, an INX B instruction will cause both registers to go to all 0s. The common pattern for the INX opcodes is 00xx 0011. A cursory examination of the opcodes reveals that the following bit patterns specify the register pairs.

Pair	Binary pattern for bits 5 and 4
BC	00
DE	01
HL	10
SP	11

This 2-bit code for the double-length registers also has a common feature with the 3-bit codes used to identify the 8-bit registers. The codes for BC, DE, and HL have the same values as the high 2-bits in the 3-bit register codes you saw earlier. For example; the 3-bit code for B is 000, while C is 001. The code for the BC pair is 00—the same as the high two bits. Like the 3-bit register code, these 2-bit codes occur in many of the 8085 opcodes.

So that you will understand all this better, let's look at a program segment to add two 4-byte numbers. The following sequence will do just that. This assumes that one 4-byte number is in registers BC and DE with the highest-order byte in B and the lowest-order byte in E. The other number is in four sequential addresses, with the lowest-order byte in the address currently indicated by HL. We also have to use instructions such as MOV A,E to move (copy) the value from E into A. The result is stored in memory, writing over the value that was there. This is done with the instruction MOV M,A (copy A to M). The MOV instructions are explained completely in Unit 3, but we must use them here to keep the program segment correct.

1)	MOV	A,E	;Copy E (low byte) to A
2)	ADD	M	;Add low bytes, do not include carry
3)	MOV	M,A	;Store result in Memory
4)	INX	H	;Point to 2nd byte
5)	MOV	A,D	;Copy D (2nd byte) to A
6)	ADC	M	;Add 2nd bytes, with carry
7)	MOV	M,A	;Store result in Memory
8)	INX	H	;Point to 3rd byte
9)	MOV	A,C	;Copy C (3rd byte) to A
10)	ADC	M	;Add 3rd bytes, with carry
11)	MOV	M,A	;Store result in Memory
12)	INX	H	;Point to 4th byte
13)	MOV	A,B	;Copy B (4th byte) to A
14)	ADC	M	;Add 4th bytes, with carry
15)	MOV	M,A	;Store result in Memory

**Figure 2-5**  
Program segment to add four bytes.

Notice that you must start with the lowest byte, because the carry from that operation affects the next operation. For example, if there is a carry from the addition of the lowest bytes, the ADC operation at line 6 will add that carry to the second two bytes. Similarly, the carry (if any) is included at lines 10 and 14. You do not include the carry at line 2 because you do not want a carry from the previous operation (whatever that was) to affect the sum of the first two bytes.



The DAD instruction (opcode 00xx 1001<sub>2</sub>) adds the specified double byte register to the HL register. This is another way the HL register is different from BC and DE. Although the DAD instructions may generate a carry, you cannot use that carry in a second DAD instruction. Therefore, DAD is limited to adding two bytes, and the primary use for the carry is to indicate that an overflow has taken place, and that the value in HL is not complete. Although you can add additional bytes with the ADC instructions, this may not be the most efficient method.

## Subtract

The subtract instructions subtract one value from another. In general, this will decrease the value of the accumulator, just as the add instructions increase it. The decrement instructions, which are also in this category have the ability to subtract one (1) from any specific register. There is no double subtract instruction, but there are two compare instructions, so there are eight subtract instructions. They have the following mnemonics and affects on the flags:

<u>Mnemonic</u>	<u>Flag Impact</u>
SUI	all flags affected
SUB	all flags affected
SBI	all flags affected
SBB	all flags affected
DCR	all except CARRY affected
DCX	no flags affected
CMP	all flags affected
CPI	all flags affected

Let's look at the SUB and SBB instructions first. The common pattern for these opcodes is 1001 xxxx<sub>2</sub>. Just like the add instructions, bit 3 determines whether or not the carry flag is considered. However, the mnemonic uses a B for borrow instead of a C for Carry. When the opcode is executed, a borrow will set the carry flag. There is not a different flag to indicate a borrow rather than a carry. Consider the following rules and examples.

1.  $0 - 0 = 0$
2.  $1 - 1 = 0$
3.  $1 - 0 = 1$
4.  $0 - 1 = 1$  with a borrow of 1.

Borrow	11111 111	11111 111	1111 111
Minuend	0000 0000	1010 1010	1000 0000
Subtrahend	<u>-0000 0001</u>	<u>-1010 1011</u>	<u>-0000 0001</u>
Difference	11111 1111	11111 1111	0111 1111

The **minuend** is in the accumulator and the **subtrahend** is identified by the opcode. In the first two examples, the subtrahend is one greater than the minuend. The result is a value which is all 1s and a borrow assigned to the carry flag. In the third example, the subtrahend is less than the minuend. Even though borrows are required for several intermediate bits, the overall result does not include a borrow.

Just as you can add A to itself, you can also subtract A from itself. You may think this is a strange instruction, since the result is always zero. However, SUB A may be preferred to MVI A,0 (move immediate 0 into A) for two reasons. First, after SUB A, the flags will be in a known condition. MVI A,0 does not affect the flags. Second, MVI A,0 requires two bytes, SUB A only one.

Because of the number of single byte registers, the most plentiful subtraction opcodes are SUB and SBB. Study the following list of opcodes. Compare it to the list of ADD and ADC opcodes, and notice the similarities and differences.

<u>Mnemonic</u>	<u>Opcode</u>	<u>Action</u>
SUB A	97	$A - A = A$
SBB A	9F	$A - A - cy = A$
SUB B	90	$A - B = A$
SBB B	98	$A - B - cy = A$
SUB C	91	$A - C = A$
SBB C	99	$A - C - cy = A$
SUB D	92	$A - D = A$
SBB D	9A	$A - D - cy = A$
SUB E	93	$A - E = A$
SBB E	9B	$A - E - cy = A$
SUB H	94	$A - H = A$
SBB H	9C	$A - H - cy = A$
SUB L	95	$A - L = A$
SBB L	9D	$A - L - cy = A$
SUB M	96	$A - M = A$
SBB M	9E	$A - M - cy = A$

The two lists (subtract and add) are very similar. The only difference in the opcodes is that bit 4 is a 1 in the subtract instructions and 0 in the add instructions. All of the operations here are minus, where they were plus in the add list. The carry (borrow) flag is still listed as cy, because it is still the carry flag even though it may test for a borrow.

The SUI and SBI instructions subtract the value given in the instruction. Again, the accumulator is the minuend, but the subtrahend is the next byte in memory after the opcode. This means that SUI and SBI are two byte instructions. In all other respects, SUI is the same as SUB and SBI is the same as SBB.

You should use DCR and DCX when you want to subtract one from the value in a register. Use DCR when you want to specify a one byte register and DCX when the register pair is BC, DE, HL, or the stack pointer. Because these are one byte instructions, you will find they work well when values up to 4 need to be subtracted. To do that, you simply repeat the DCR or DCX instruction the required number of times.

These instructions compare the byte in the accumulator with another byte specified by the opcode. In the case of `CMP`, it is one of the registers or the `M` address specified by the `HL` index. `CPI` uses the value in memory immediately after the `CPI` opcode. These instructions are listed with `SUB` and `SUI` because their effect on the flags is the same. In other words, the flags will be the same after you do a `CMP B` as they would be after a `SUB B`. It is therefore very helpful to think of them in the same category.

As mentioned earlier, there is not a double-length subtract instruction. Subtracting from a value in `HL` is not as easy as adding to it. It is also difficult to complement the value in `HL`, so subtraction by addition is complicated.

Now answer the following self-test review questions and then perform Experiment 4.

### Self-Test Review

11. The result of an `ADD E` instruction is placed in the \_\_\_\_\_.
12. `DAD B` adds the \_\_\_\_\_ register pair to the `HL` register pair and puts the result in the \_\_\_\_\_ register pair and the carry flag.
13. The `INR` instruction affects all flags except the \_\_\_\_\_ flag.
14. `INR` and `DCR` instructions affect \_\_\_\_\_-byte registers.
15. `INX` and `DCX` instructions affect \_\_\_\_\_-byte registers.
16. The `X` in `INX` and `DCX` means \_\_\_\_\_.
17. The \_\_\_\_\_ register pair always points to a byte identified as `M` in the instruction mnemonics.
18. The `I` in `ADI`, `ACI`, `SUI`, and `SBI` stands for \_\_\_\_\_.
19. `CMP` and `CPI` affect the flags the same as a/an \_\_\_\_\_ instruction.

## SPECIAL ARITHMETIC AND LOGIC OPCODES

Besides addition and subtraction, the 8085 is also capable of several logical combinations of two values. From your experience with digital circuits, you should be familiar with the expressions AND, OR, and Exclusive OR. As you also know, the circuits in a digital computer are based on these logic combinations. It should seem natural therefore, that the microprocessor allows you to use these functions to combine values within a program. For simplicity, these three operations will be considered separately, although they have much in common.

One thing they all have in common is that they affect all the flags.

### Logical AND.

When you AND two bytes, the individual bits of the byte are considered separately. For example, if you AND the values 1010 1111<sub>2</sub> and 1111 0101<sub>2</sub>, you get 1010 0101<sub>2</sub>. The carry flag will always be cleared after an AND operation. Let's review the AND function quickly before you continue.

The logic table in Figure 2-6 shows the possible combinations of two bits. The logic operations in the 8085 will not combine more than two bits during any operation, so this table is sufficient.

<u>bit A</u>	<u>bit B</u>	<u>A AND B</u>
0	0	0
0	1	0
1	0	0
1	1	1

**Figure 2-6**  
AND logic table.

The rule is that the result is only a 1 when both inputs are 1's. Another way to say this is that any 0 input will result in a 0 output.

The mnemonics for the 8085 AND operations are ANA and ANI. After the ANA, you indicate the 1-byte register A, B, C, D, E, H, L, or M. If you use ANI, you specify the value to be ANDed. The opcode for the ANA instructions have the pattern 1010 0xxx<sub>2</sub>. Bits 0, 1, and 2 indicate the 1-byte register that will be ANDed with the accumulator. As usual, the result will be placed in the accumulator, and all flags are affected. ANA A can be used to check the value in the accumulator without changing it. This may be useful if the value was moved to A from some other location.

The ANI instruction is like the other immediate instructions. The operand byte is in the next address. Other than that, it is identical to the ANA instruction. All flags are affected, and the carry flag will be 0 after an ANI opcode is executed. ANI is also useful when you want to clear all but certain bits in a byte. For example, if you want to know if bit 0 is a 1, you can use ANI 01 to clear all but bit 0. Then, if the value in the accumulator is zero, you know that bit 0 is a 0. If the byte is not zero, bit 0 must be a 1. This operation is called **masking**, because all 0 values in the mask will be 0's in the result. The only bits that will be 1's are those that are 1's in both the accumulator and the operand. You will learn more about this under the heading of flags in Unit 4, but Figure 2-7 will show you how an AND logic mask works.

test byte	0100 0011	bits 0, 1, & 6 are set
AND mask	0000 0010	to test bit 1
result	0000 0010	only bit 1 is set
test byte	0100 1001	bits 0, 3, & 6 are set
AND mask	0000 0010	to test bit 1
result	0000 0000	no bits are set

**Figure 2-7**  
AND logic mask.

In the first example, bit 1 is set (1) in both the test byte and the mask. As a result, the AND function has bit 1 set, so the value is NOT zero. In the second example, bit 1 is not set in the test byte, so the result is zero.

The purpose of this course is to teach you the instructions for the 8085, so we cannot show you all the ways you can use the instructions. This is left to you and your imagination. However, you must keep in mind the rules for each operation.

## Logical OR

When you OR two bytes, the result will have 1's where either of the source bytes contained 1's. The table in Figure 2-8 shows the result of combining the four different possible inputs. For OR operations the rule is that the output is a 1 whenever either input is a 1. Or, the output is 0 only when both inputs are 0. The carry flag and auxiliary carry flag will always be cleared after an OR function.

<u>bit A</u>	<u>bit B</u>	<u>A OR B</u>
0	0	0
0	1	1
1	0	1
1	1	1

**Figure 2-8**  
OR logic table.

The mnemonics for 8085 OR operations are ORA and ORI. The ORA instructions are followed by the name of the 1-byte register that you want to OR with the accumulator. The opcode for ORA is 1011 0xxx<sub>2</sub>, which is the same as ANA except bit four is now a 1. Like ANI, ORI can also be used to test for a specific bit. If you wish to test bit 0, you can apply ORI FE<sub>16</sub> and check if the resulting byte has an odd or even number of 1s. An even number of 1s means bit 0 is a 1, and an odd number of 1s means bit 0 is a 0. This is also a mask, but this time the values that are 1s in the mask will be 1s in the result. Again, you will learn more about masks under the heading of flags in Unit 4, but Figure 2-9 does show you how an OR logic mask works.

test byte	0100 0011	bits 0, 1, & 6 are set
OR mask	1111 1101	to test bit 1
result	1111 1111	number of 1s is even
test byte	0100 1001	bits 0, 3, & 6 are set
OR mask	1111 1101	to test bit 1
result	1111 1101	number of 1s is odd

**Figure 2-9**  
OR logic mask.

In the first example, bit 1 in the test byte is being tested to see if it is set (1). Therefore, all of the bits in the mask are set except for bit 1. Because bit 1 in the test byte is set, the OR function causes all bits to be set in the result. This produces in an even number (8) of 1s in the result, which will be reflected in the parity flag. In the second example, bit 1 is not set in the test byte, so it is not set in the result. Therefore, an odd number of bits (7) are 1s in the result. This is only one of the many ways you can use the OR instruction. It is an interesting fact that ORA A and ANA A have the same result. The accumulator is unchanged, but the flags are set or cleared appropriately.



## Logical Exclusive OR

The rule for the exclusive OR (XOR) function is that the output is a 1 when the two inputs are different. This is the same as saying that the output is a 0 whenever both inputs are the same. This is useful when you want to selectively change one or more of the bits in a byte. Figure 2-10 is a table showing the input and output values for the XOR function.

<u>bit A</u>	<u>bit B</u>	<u>A XOR B</u>
0	0	0
0	1	1
1	0	1
1	1	0

**Figure 2-10**  
XOR logic table.

The 8085 exclusive OR instructions are XRA and XRI. By now, you should be getting used to the structure of these mnemonics, so it should be obvious that XRA is followed by the 1-byte register name, which will be exclusively ORed with the accumulator. By referring to the Opcode Chart you can see that the opcode for the XRA instructions is 1010 1xxx. XRI, of course, combines the next byte in memory with the value in the accumulator. Like the AND and OR functions, the carry flag and auxiliary carry flag will always be cleared after an OR function. The one byte instruction XRA A is sometimes used to clear the accumulator.

Figure 2-11 shows the effect of using an exclusive OR mask on a byte. For this example, we have used the output of the first operation as the input of the second, you can see that the byte returns to its original value when the mask is applied twice.

test byte	0100 0011	bits 0, 1, & 6 are set
XOR mask	0000 1111	toggle the low nibble
result	0100 1100	high nibble same, low nibble complimented
test byte	0100 1100	bits 2, 3, & 6 are set
XOR mask	0000 1111	toggle low nibble
result	0100 0011	high nibble same, low nibble complimented

**Figure 2-11**  
XOR logic mask.

Any 1 in the XOR mask toggles the corresponding bit in the test byte. One use for this is to convert lowercase ASCII characters to uppercase and vice versa. The difference between uppercase and lowercase ASCII characters is that bit 6 is a 1 for lowercase and a 0 for upper case. If you use an XOR mask of 0100 0000<sub>2</sub>, bit 6 will toggle, making lowercase upper and uppercase lower.

Think about that and decide how you might use the AND and/or OR functions to covert all characters to uppercase or all to lowercase.

Have you thought about it? To convert uppercase letters to lowercase, you must change bit 6 from a 0 to a 1. This is easily done by ORing a byte that has only bit 6 set, with the ASCII character. In Experiment 5, you will have an opportunity to see this, and then exercise the instructions that convert lowercase to uppercase.

Now answer the following self-test review questions and then perform Experiment 5.

## Self-Test Review

20. The mnemonic for the instruction that ANDs the next (immediate) byte with the accumulator is\_\_\_\_\_.
21. When you AND  $1011\ 1111_2$  with  $0110\ 0001_2$  the result is  
\_\_\_\_\_.
22. The opcode for ORI is\_\_\_\_\_ (Refer to Figure 2-4 if necessary.)
23. If you want to AND the accumulator with the H register the mnemonic to use is\_\_\_\_\_.
24. What will be the value in the accumulator after XRA A? \_\_\_\_\_.
25. An AND mask has\_\_\_\_\_ 's in the bits you want to retain.
26. An exclusive OR mask has\_\_\_\_\_ 's in the bits you wish to remain the same.
27. What is the mnemonic to AND the contents of memory (as pointed to by HL) with the accumulator.\_\_\_\_\_.

## SHIFT AND OTHER LOGIC OPERATIONS

The remaining instructions and opcodes for this unit are listed in Figure 2-12. These fall into three categories. The first category contains the instructions that rotate, or shift, the accumulator. The second has only one instruction, which compliments the accumulator. And the last category, also with one instruction adjusts the accumulator for binary coded decimal.

Mnemonic	Opcode	Flags Affected	Action
RLC	07	carry only	Rotate A left, $A_7 = A_0$ , $A_7 = cy$
RRC	0F	carry only	Rotate A right, $A_0 = A_7$ , $A_7 = cy$
RAL	17	carry only	Rotate A left, $A_7 = cy$ , $cy = A_0$
RAR	1F	carry only	Rotate A right, $A_0 = cy$ , $cy = A_7$
CMA	2F	none	A compliment = A
DAA	27	all	Decimal adjust A = A

**Figure 2-12**  
Rotates, DAA, and CMA.

### DAA and CMA

Let's look at the last two categories first. DAA, opcode 0010 0111<sub>2</sub>, adjusts the accumulator so that the value is in a format called binary coded decimal, or BCD. Binary coded decimal means that instead of the eight bits representing a number between 0 and 255<sub>10</sub>, they represent two decimal digits. This allows a range between 0 and 99<sub>10</sub>. Figure 2-13 is a table of the BCD values.

Because BCD is a convenient notation, it is often used to represent numbers that must be displayed. It requires special handling, as you will see, and requires more space in memory than a simple binary number.

<u>Decimal</u>	<u>Binary</u>	<u>BCD</u>
0	0000	0000 0000
1	0001	0000 0001
2	0010	0000 0010
3	0011	0000 0011
4	0100	0000 0100
5	0101	0000 0101
6	0110	0000 0110
7	0111	0000 0111
8	1000	0000 1000
9	1001	0000 1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101

**Figure 2-13**  
Binary Coded Decimal.

From 0 through 9, you should recognize these as the same as the hexadecimal values. For values greater than 9, the BCD requires a second nibble that is not all zeros.

The microprocessor adds two bytes in the same way, regardless of whether the code is BCD or not. However, if the values are BCD this could lead to some errors. The examples in Figure 2-14 show what can happen.

carry	0110 100	1000 1001	10000 001	0001 000
	0101 0101	0101 0110	1001 1000	0000 1000
	<u>+0011 0100</u>	<u>+0100 0100</u>	<u>+1000 0001</u>	<u>+0000 1000</u>
total	1000 1001	1001 1010	10001 1001	0001 0000
	55	5 6	98	08
	<u>+34</u>	<u>+4 4</u>	<u>+81</u>	<u>+08</u>
	89	910	119	10

**Figure 2-14**  
BCD addition.

The first example (in the left column) adds the binary values for  $85_{10}$  and  $52_{10}$  and produces the sum  $137_{10}$ . If the values were BCD,  $55_{10}$  plus  $34_{10}$  produces a total of  $89_{10}$ , which is also correct.

In the second example, the situation is slightly different. The binary addition sees the values as  $86_{10}$  plus  $68_{10}$  for a total of  $154_{10}$ . However, if the numbers are BCD, the addition is  $56_{10}$  plus  $44_{10}$  and a total of  $9 \underline{10}_{10}$ . But that cannot be! The highest nibble in BCD is equal to  $9_{10}$ . A nibble value of  $10_{10}$  is not allowed. Obviously an adjustment is needed.

Let's check the third example. In binary, the addition is  $152_{10}$  plus  $129_{10}$  for a total of  $281_{10}$  and a carry. But in BCD, the addition would be  $98_{10}$  plus  $81_{10}$ , for a total of  $179_{10}$  plus a carry. Obviously an error.

In the fourth example, the numbers are very simple. In standard binary the values represent  $8_{10}$  plus  $8_{10}$ , for a total of  $16_{10}$ . In BCD it is also  $8_{10}$  plus  $8_{10}$ , but the total only represents  $10_{10}$ . How can this problem be corrected?

The answer is to use the decimal adjust accumulator (DAA) instruction. DAA makes use of the carry flag and an intermediate, or auxiliary, carry flag (A or AC). The A flag is set whenever there is a carry from bit 3 to bit 4 during a math operation in the accumulator.

Let's see how this can help solve the problem. In the first example, no adjustment was necessary. It is also obvious that neither the carry nor the A flag would be set. In the second example, neither the carry nor the A flag are set, but there is a problem. It appears that the lower nibble is greater than 9. To correct the problem use the first DAA rule is as follows:

1. *When the resulting low nibble value is greater than 9, DAA must add 6 to the result.*

Let's see how that looks:

carry	1000 100
	0101 0110
	<u>+0100 0100</u>
subtotal	1001 1010
add 6	<u>+0000 0110</u>
total	1010 0000

But wait, the BCD now says  $56_{10}$  plus  $44_{10}$  is  $100_{10}$ . At first glance, you might think that is right, but remember, BCD does not permit any nibble with a value greater than 9. This leads to the second rule:

2. *If the result in the high nibble is greater than 9, DAA must add 6 to it—that is, add  $60_{10}$  to the byte value.*

Let's check example 2 again.

carry	1000 100	
	0101 0110	
	<u>+0100 0100</u>	
subtotal	1001 1010	
add 6	<u>+0000 0110</u>	(rule 1)
subtotal	1010 0000	
add 60	<u>+0110 0000</u>	(rule 2)
total	1 0000 0000	

Now the value is correct.  $56_{10}$  plus  $44_{10}$  is  $00_{10}$  plus a carry to the next byte ( $100_{10}$ ).

But what about the third example? Neither result is greater than 9, but the high nibble is only one, and it should be seven. This leads to a modification of the second rule:

2. *(Revised) If the result in the high nibble is greater than 9, or the carry flag is set, DAA must add 6 to it—that is, add  $60_{10}$  to the byte value.*

Using revised second rule, the third example is now correct— $98_{10}$  plus  $81_{10}$  equals  $79_{10}$  plus a carry.

In the fourth example, the carry is not set, and no nibble is greater than 9. This is why the A flag is needed. Notice that there is a carry from the low nibble to the high nibble. This causes the high nibble to be a 1, which is correct, but it also indicates that the low nibble is incorrect. Again, DAA must add 6, so we will restate the first rule as follows:

*1. (Revised) When the resulting low nibble value is greater than 9, or the A flag is set, DAA must add 6 to the result.*

With this change, the fourth example correctly states that  $8_{10}$  plus  $8_{10}$  equals  $16_{10}$ .

Fortunately, it is not normally necessary to remember the rules the DAA instruction uses. Just remember to use DAA after any addition operation on BCD numbers.

The last instruction to be described in this category is CMA (**complement the accumulator**). You can use this instruction any time you want to change all the bits in the accumulator. Any that were 1s will be 0s and those that were 0s will become 1s. None of the flags are affected by the CMA instruction. From the opcode chart you can see that the opcode for CMA is  $0010\ 1111_2$ . The difference between the CMA instruction and the XRI instruction, written as  $XRI\ FF_{16}$ , is how the flags are affected and the fact that CMA is a single-byte instruction.

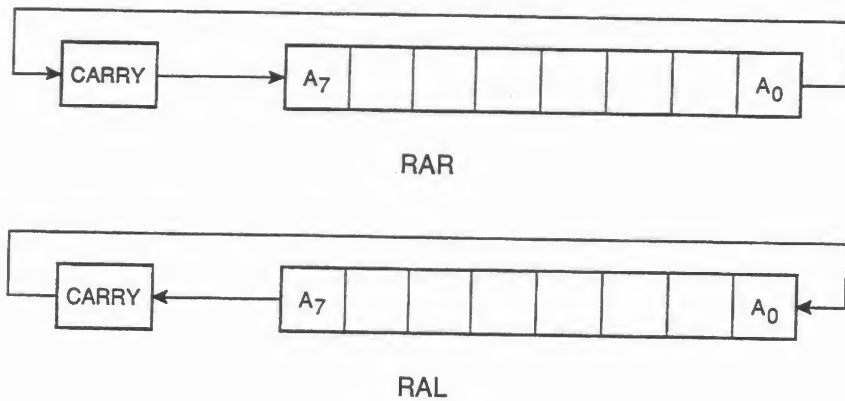
## Shifts or Rotates

The shift, or rotate, instructions cause the bits in the accumulator to be transferred into their neighbor on either the right or the left, depending on the instruction. Although the carry flag is involved in all rotation instructions, it can be used in two different ways.

In the RAR and RAL instructions, the carry flag is included as though it were a ninth bit of the register. Therefore, these are actually nine bit rotations. Some have called these **rotate around right** (RAR) and **rotate around left** (RAL) to distinguish them from the other rotation instructions. RAR is a quick way to divide the accumulator by two. The remainder is in the carry flag. If the carry flag is zero, RAL effectively multiplies the accumulator by two. However, it is no faster than ADD A, which also multiplies it by two.



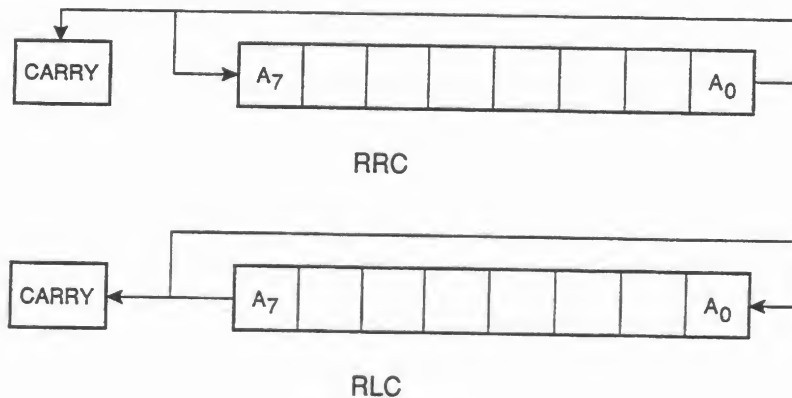
Also, you must clear the carry before you can use RAR or RAL to multiply or divide a second time. RAR and RAL are illustrated in Figure 2-15.



**Figure 2-15**  
How the RAR and RAL instructions work.

The RAR instruction shifts all bits to the right. The value that was in  $A_0$  is transferred to the carry flag, and the value that was in the carry flag is transferred to  $A_7$ . The RAL instruction shifts all bits to the left. The value that was in  $A_7$  is transferred to the carry flag, and the value that was in the carry flag is transferred to  $A_0$ .

You should remember RRC as **rotate right without the carry** and RLC as **rotate left without the carry**. These are eight bit rotations. The carry flag is the same as the bit that was rotated out of the end of the register. These two instructions are illustrated in Figure 2-16.



**Figure 2-16**  
How the RRC and RLC instructions work.

The RRC instruction shifts all bits to the right. The value that was in A<sub>0</sub> is transferred to A<sub>7</sub> and copied to the carry flag. The RLC instruction shifts all bits to the left. The value that was in A<sub>7</sub> is transferred to A<sub>0</sub> and copied to the carry flag. In both these instructions, the value that was previously in the carry flag is lost.

Now answer the following self-test review questions and then perform Experiment 6.

### Self-Test Review

28. After an RRC instruction, the carry flag will be the same as bit \_\_\_\_.
29. The RAR instruction causes the value in the bit A<sub>7</sub> to be transferred to \_\_\_\_\_.
30. The DAA instruction adds 6 when the low nibble is greater than 9, or the A flag is set.
31. The DAA instruction adds \_\_\_\_ to the high nibble when the high nibble is greater than \_\_\_\_, or the \_\_\_\_\_ flag is set.
32. Which instruction divides the accumulator by two, leaving the remainder in the carry flag? \_\_\_\_\_.
33. Which flags are affected by the CMA instruction? None.

## UNIT SUMMARY

1. The 8085 has 10 accessible registers: the accumulator, the flags, B, C, D, E, H, L, the stack pointer (SP), and the program counter (PC).
2. The accumulator, the flags, B, C, D, E, H, and L are 8-bit registers.
3. The program counter and the stack pointer are 16-bit registers.
4. The accumulator and flag registers can be accessed as a 16-bit register called the processor status word (PSW).
5. B & C, D & E, and H & L form register pairs that are identified by the name of the high order byte (i.e. B, D, and H).
6. The 16-bit registers B, D, H, and SP form a group called the register array.
7. The flags are sign (S), zero (Z), auxiliary carry (A or AC), parity (P), and carry (C or CY).
8. The A register is the same as the accumulator.
9. Don't confuse the A register and the A flag.
10. Don't confuse the C register and the C flag.
11. There are 246 usable opcodes in the 8085.
12. The 8-bit registers are identified by three bits of the opcode as follows:

<u>Register</u>	<u>Binary pattern</u>
A	1 1 1
B	0 0 0
C	0 0 1
D	0 1 0
E	0 1 1
H	1 0 0
L	1 0 1
M	1 1 0

13. The 16-bit register pairs are identified by two bits of the opcode as follows:

<u>Register</u>	<u>Binary pattern</u>
B	00
D	01
H	10
SP	11

14. The addition mnemonics are ADD r, ADC r, ADI, and ACI. The "r" is replaced by the name of an 8-bit register (A, B, C, D, E, H, or L) or M for memory. The mnemonics ADC and ACI add one if the carry flag is set (1). The mnemonics ADI and ACI use the next byte as an immediate operand.
15. DAD r adds the specified register pair to the HL pair.
16. The subtraction mnemonics are SUB r, SBB r, SUI and SBI. The r is replaced by the name of an 8-bit register (A, B, C, D, E, H, or L) or M for memory. The mnemonics SBB and SBI subtract one if the carry flag is set (1). The mnemonics that have an I in them use the next byte as an immediate operand.
17. INR r adds 1 to the identified byte.
18. INX r adds 1 to the identified 16-bit register pair.
19. DCR r subtracts 1 from the identified byte.
20. DCX r subtracts 1 from the identified 16-bit register pair.
21. ANA r and ANI logically AND the specified byte with the accumulator.
22. ORA r and ORI logically OR the specified byte with the accumulator.
23. XRA r and XRI logically exclusive OR the specified byte with the accumulator.
24. CMP r and CPI compare the specified byte with the accumulator, which sets the flags the same as if a subtract had taken place.
25. RLC rotates the bit values in the accumulator to the left, the carry will have the same value as bit A<sub>0</sub>, or old bit A<sub>7</sub>.

26. RRC rotates the bit values in the accumulator to the right, the carry will have the same value as bit A<sub>7</sub>, or old bit A<sub>0</sub>.
27. RAL rotates the bit values in the accumulator to the left, the carry will get its value from bit A<sub>7</sub>, and bit A<sub>0</sub> will get the value that was in the carry flag.
28. RAR rotates the bit values in the accumulator to the right, the carry will get its value from bit A<sub>0</sub>, and bit A<sub>7</sub> will get the value that was in the carry flag.
29. CMA causes all 1's in the accumulator to change to 0's and all 0's to become 1's.
30. DAA corrects the accumulator to binary coded decimal by the following rules.
  1. When the resulting low nibble value is greater than 9, or the A flag is set, 6 is added to the value in the accumulator.
  2. When the resulting high nibble value is greater than 9, or the C flag is set, 6 is added to the value in the high nibble.

*Unit 3*

# **ADDRESSING MODES**

## CONTENTS

INTRODUCTION . . . . .	3-3
UNIT OBJECTIVES . . . . .	3-4
MOVE INSTRUCTIONS . . . . .	3-5
IMMEDIATE ADDRESSING . . . . .	3-7
Assembly Language . . . . .	3-7
Immediate Addressing to 16-bit Registers . . . . .	3-8
STORES AND LOADS . . . . .	3-9
Indirect Loads and Stores . . . . .	3-9
Direct Loads and Stores . . . . .	3-10
OTHER REGISTER TRANSFERS . . . . .	3-12
UNIT SUMMARY . . . . .	3-15
EXPERIMENTS . . . . .	3-17

## INTRODUCTION

Addressing modes are the ways a microprocessor accesses data. Each microprocessor has its own methods. If you have previously studied Motorola microprocessors, you will find the addressing of the 8085 somewhat different. The 8085 has three addressing modes: direct, indirect, and immediate. In Unit 2, you saw two of these ways that the 8085 can access data: indirect and immediate. You will recognize these as they are discussed.

Also, because this unit focuses on the movement of data, you will see that the addressing modes are closer related to data movement than they are to arithmetic and logic operations.

The addressing mode is only one way of classifying the instruction set. Another way, as you learned in Unit 2 is by the type of operation performed. A third way is by the number of bytes required for the instruction.

Most of the 8085 instructions accomplish their work with a single byte. When an addressing mode is involved, all single byte instructions use indirect addressing. In indirect addressing, a register has the address of the data. Two byte instructions use immediate addressing. In immediate addressing, the operand (the second byte) is the data. All two or three byte 8085 instructions involve some addressing mode. Three byte addressing mode instructions can use either direct or immediate addressing. That is, the second and third byte may contain the data (immediate addressing), or they may be the address of the data (direct addressing).

All 8085 addressing is absolute. The value given by the register or operand is the specific address in memory, and it is not affected by the location of the program. In many other microprocessors, the addressing may be relative to the opcode or to a certain other memory address.

Again, you will find it helpful to refer to the table of opcodes as you read this unit.



## UNIT OBJECTIVES

When you complete this unit you will be able to:

1. Identify the type of addressing used by each 8085 move, load, or store instruction.
2. Define the three 8085 addressing modes.
3. Identify and explain the effect of the 8085 exchange instructions.
4. Given a desired data transfer, select the correct move, load, store, or exchange instruction to perform that data transfer.
5. Define the term label.
6. State the criteria an assembler uses to determine if the operand is a number or a label.
7. Identify the assembler notations for decimal, binary, and hexadecimal.

## MOVE INSTRUCTIONS

The most common 8085 instruction is the move instruction. The standard **MOV** instruction involves a single byte opcode, and the registers involved are identified by the mnemonic. Most of these are identified by the mnemonic **MOV r1,r2** -- where r1 is the destination and r2 is the source. The destination register is over written by the value in the source. This is illustrated in Figure 3-1. Figure 3-1A shows the registers before the **MOV A,B** instruction. For illustration, the registers are filled with the hexadecimal values 11, 22, ..., CC. Therefore A has a beginning value of 11<sub>16</sub>, and B has a beginning value of 33<sub>16</sub>.

This is illustrated in Figure 3-1. Figure 3-1A shows the registers before the **MOV A,B** instruction. For illustration, the registers are filled with the hexadecimal values 11, 22, ..., CC. Therefore A has a beginning value of 11<sub>16</sub>, and B has a beginning value of 33<sub>16</sub>.

		S Z X AC X P X C															
A	00010001								00100010								FLAGS
B	0	0	1	1	0	0	1	1	0	1	0	0	0	1	0	0	C
D	0	1	0	1	0	1	0	1	0	1	1	0	0	1	1	0	E
H	0	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	L
PC	1	0	0	1	1	0	0	1	1	0	1	0	1	0	1	0	
SP	1	0	1	1	1	0	1	1	1	1	0	0	1	1	0	0	

Figure 3-1A

																S Z X ACX P X C															
A	00110011								00100010								FLAGS														
B	0	0	1	1	0	0	1	1	0	1	0	0	0	1	0	0	C														
D	0	1	0	1	0	1	0	1	0	1	1	0	0	1	1	0	E														
H	0	1	1	1	0	1	1	1	1	0	0	0	1	0	0	0	L														
PC	1	0	0	1	1	0	0	1	1	0	1	0	1	0	1	1															
SP	1	0	1	1	1	0	1	1	1	1	0	0	1	1	0	0															

Figure 3-1B

The MOV instruction.

Figure 3-1B shows the registers after the **MOV A,B** instruction. Only two values have changed. The only one that is important to this discussion is the accumulator, or A register. It now contains the same value that was

in the B register. The B register remains unchanged. Also, notice that the flags remain unchanged, because the move instructions do not affect the flags. The other register that changed is the program counter. It has been incremented by one to point to the next instruction.

Instructions in this form involve no specific addressing mode, because the two registers involved contain the data. However, fourteen MOV instructions use indirect addressing. These instructions have two forms MOV M,r and MOV r,M. In both cases, the r identifies one of the 8085's 1-byte registers (A, B, C, D, E, H, or L). However, the 8085 does not have an M register. The M identifies the byte of data indexed, or pointed to, by the H-L register pair. This is indirect addressing, because the register (H-L) contains the address of the data.

Together, the MOV instructions account for 63 of the 8085's opcodes. This is more than one-fourth of the microprocessor's executable opcodes. The MOV instructions are the easiest way to transfer a byte from memory to a register, or from a register to a selected address in memory.

## IMMEDIATE ADDRESSING

Closely associated with the MOV instructions are the MVI instructions. The only difference is that MVI instructions include a number instead of a source byte. Thus, the form of the instruction is MVI r,byte. The value of the byte is immediately moved to the register specified. However, one of the MVI instructions can be considered both an indirect and an immediate addressing instruction. The source is the immediate byte, but the destination is the address M. M, as you will remember, is selected by the address in the H-L register pair.

### Assembly Language

The mnemonics are used not only for written notation of the machine code, but also to be translated, or assembled, into machine code by a program called an assembler. To do this, the assembler program must read the mnemonics and translate them into machine code. In the case of the mnemonics themselves, this requires only that they be spelled exactly right. But in the case of numbers, there is a problem. In many cases, it is helpful to specify numbers as binary or hexadecimal, instead of decimal. The most common convention is to add a letter to the end of the number. B is used for binary, and H is used for hexadecimal. The default values (no letter) are decimal.

A second difficulty occurs, because assembly language allows the use of labels. **Labels** are groups of letters that stand for an address or number. The difficulty in allowing labels is that it then becomes impossible for the assembler to tell the difference between numbers and labels. To resolve this difficulty, the following convention has been adopted.

1. All numbers must begin with a numeral.
2. All labels must begin with a letter.

For decimal and binary numbers, this raises no problems, because all binary and decimal values are represented by numerals. For hexadecimal values, the convention frequently requires the addition of a leading zero. Therefore, if the number is 33<sub>10</sub>, it can be represented as 33, 00100001B, or 021H. Although it is not necessary, it is customary to include the leading zeros. However, for numbers 160<sub>10</sub> or greater, you must include the zero in the hexadecimal notation (0A0H). The zero tells the assembler that this is a number, not a label.

Let's look at an example of how this affects the MVI instruction. First assume that you want to put a value of 5 into the accumulator. The mnemonic for this is MVI A,5. There is no ambiguity in the number 5, because it is the same in decimal and in hexadecimal. Next, assume you want to put a value of 188<sub>10</sub> into the accumulator. The normal way to do this is with the instruction MVI A,188. This specifies the decimal value and the assembler makes the conversion to binary. However, if you wanted to apply a mask of 10111100<sub>2</sub> it is easier to represent the number in hexadecimal -- MVI A,0BCH. Although a very experienced programmer might recognize this as 188<sub>10</sub>, most would have to figure it out. It is easier to have the assembler program do this for you.

### Immediate Addressing to 16-bit Registers

Just as you might want to put an 8-bit number into a 1-byte register, you might also want to put a 16-bit number into a 2- byte register, or register pair. Because the register pairs are identified by only the letter of the high order byte, a different mnemonic was chosen -- **LXI**.

For example, if you want to put a value of 30,000<sub>10</sub> into the D-E pair, you would use the instruction LXI D,30000. If you use an LXI instruction and a value less than 256, the high byte will be zero. For example, LXI B,10 will put a zero into B and a 10 into C. In Unit 2, you learned that DAD, INX, and DCX could be applied to B, D, H, and SP. This is also true of LXI. You cannot use LXI for the program counter or the PSW. You will learn how to change those registers in Unit 5.

## STORES AND LOADS

The opcodes in the form 00xx xx10 are identified by mnemonics that indicate they either store values, or transfer the value to memory, or load values, by bringing them from memory. These instructions are either direct or indirect as indicated in Figure 3-2.

<u>Mnemonic</u>	<u>Opcode</u>	<u>Mode</u>	<u>Action</u>
STAX B	02	Indirect	A => Address B
LDAX B	0A	Indirect	Address B => A
STAX D	12	Indirect	A => Address D
LDAX D	1A	Indirect	Address D => A
SHLD adr	22	Direct	L => adr, H => adr+1
LHLD adr	2A	Direct	adr+1 => H, adr => L
STA adr	32	Direct	A => adr
LDA adr	3A	Direct	adr => A

**Figure 3-2**

Store and Load instructions.

Although these instructions appear to be in two categories (store and load) it is easier to discuss them in relation to their addressing modes. Four use indirect addressing, and the other four use direct addressing.

### Indirect Loads and Stores

The indirect load and store opcodes use a register pair (either B-C or D-E) to point to an address in memory. The load operations load the data from that address into the accumulator. The store operations copy the data in the accumulator to memory. The mnemonics stand for the operation to be performed.

For example, the load operations start with the letters LD, for load, and the store instructions start with the letters ST, for store. The A, of course, stands for the A register (the accumulator). The X is for index, just as it was in DCX and INX. The last letter indicates which register pair to use as an index.

You might have expected to find the instructions STAX H and LDAX H. But if you will think about it for a moment, you will realize that because the HL pair points to the address called M, STAX H is identical to MOV M,A. Similarly, LDAX H means the same as MOV A,M. There are other instructions you will learn about in Unit 5, that perform the same functions with regard to the stack pointer.

In summary, the STAX and LDAX instructions are very close in nature to the MOV r,M and MOV M,r instructions. Keep them in mind whenever you are using the BC and DE register pairs as pointers.

## Direct Loads and Stores

There are two direct loads and two direct stores. As you will recall, direct addressing means that the operand is the memory address to be used. For example, if the mnemonic is STA 5100H, the data in the accumulator will be copied into address 5100<sub>16</sub>.

Similarly, LDA 5100H will copy the data currently at address 5100<sub>16</sub> into the accumulator.

Let's compare this to the immediate instruction, MVI 10H. Notice that the load instruction can accept a 2-byte word address, but the operand in the move immediate instruction must be placed in the accumulator, which can only hold a single byte. What if you want to put two bytes into a certain pair of memory locations?

This is the function of the SHLD instruction. SHLD (Store HL Direct) and LHLD (Load HL Direct) are the only direct addressing instructions that are capable of handling two bytes of data. You must be careful when you work with these instructions. Always remember: the operand provides the address and the H and L registers provide the data. These are reciprocal operations. Data stored with SHLD can be retrieved with LHLD.

After an SHLD instruction, the operand address will have a copy of the data in the L register, and the next higher address will have a copy of the H register.

After an LHLD instruction, the data at the operand address will have been copied into the L register, and the data at the next higher address from the operand will have been copied into the H register.

**Self-Test Review**

1. In the mnemonic MOV r1,r2 the r1 is the Destination and r2 is the Source.
2. In direct addressing, the operand is the memory Address of the data to be copied.
3. In indirect addressing, the operand identifies the memory address that points to the data to be copied.
4. In immediate addressing, the operand is the \_\_\_\_\_ to be copied.
5. MVI A,21 uses \_\_\_\_\_ addressing.
6. MOV A, M uses \_\_\_\_\_ addressing.
7. What direct addressing instruction will copy the values from the H and L registers into memory? \_\_\_\_\_
8. The assembler distinguishes labels from numbers, because numbers start with a \_\_\_\_\_.
9. LDAX D is an example of \_\_\_\_\_ addressing.
10. Under what circumstance would MOV A,M be the same as LDA 5100H? \_\_\_\_\_



## OTHER REGISTER TRANSFERS

The 8085 has four instructions that quickly move the values in the register pairs around. these are valuable to you as a programmer, because they allow you to effectively use the 16-bit registers for more than one value at a time. As you know, the H-L pair point to an address called M. M is accessible for use as an operand in arithmetic and logic operations. Wouldn't it be nice if you could use D-E also?

What if you wanted to write a program that would work at any location in memory? But, all the 8085 jump instructions are to specific memory addresses. What can you do about that?

All these things, and many more can be accomplished by using the following four special register transfers:

PCHL	Copy H & L to the Program Counter
SPHL	Copy H & L to the Stack Pointer
XTHL	Exchange the top of the stack and H & L
XCHG	Exchange D & E and H & L

As you may notice, these fall into two general types: copies and exchanges. This is very important! When something is copied the old value at the destination is lost, and it cannot be regained. When values are exchanged, both values can still be used.

The two copy instructions allow you to put the value currently in H & L into either the program counter or the stack pointer. For example, if you copy HL into the program counter, that will become the address of the next instruction to be executed. If you copy H & L to the stack pointer, that address will become the top of the stack.

The stack is an area in memory that you can use for temporary storage of the registers. You will learn how to use the stack in Unit 5. For now, all you need to know is that there are special instructions relating H & L to the stack pointer and the stack.

The letters in the mnemonic represent the operation. PC means program counter, SP means stack pointer, and HL identifies the HL register pair. Therefore, SPHL means to move the value in HL into the SP register. Notice that like the MOV instructions, the destination is first and the source is second. PCHL means to move the contents of HL into the PC register.

The SPHL instruction is only useful to initialize, or set up a stack. There is no provision in the 8085 for saving the old stack pointer value. It is forever lost when the SPHL instruction is executed. Because of this, you must use SPHL with extreme care and much thought. Do not use it casually.

Similarly, PCHL is a "brute force" jump. You cannot save the program counter, so there is no way to return to that point. PCHL is occasionally called an "indirect" or "computed" jump. Like SPHL, PCHL must be used with care.

The exchange instructions are much more useful, because they preserve both values. In other words, if you repeat the instruction, and have not modified either value the contents will be the same. You will probably find that XCHG is the more useful operation. It allows you to use the D & E register pair almost as easily as you use the H & L pair. Here is how. If you write a routine that works with M to perform a given task, you can assign the memory location for one set of figures to H & L and the location for a second set to D & E. Then, after you use the routine with H & L, you do an XCHG and repeat the routine, using exactly the same instructions. Again, the letters in the mnemonics identify the operations. XCHG simply means "exchange." You must remember what is exchanged.

XTHL is a more conventional notation. The X means exchange, T stands for top, and HL indicates the HL register pair. The XTHL instruction exchanges the H & L registers with the top of the stack. The top of the stack is always defined as the address pointed to by the stack pointer. Therefore, when you execute XTHL the following operations are performed:

$$\begin{aligned} (L) &\longleftrightarrow ((SP)) \\ (H) &\longleftrightarrow ((SP)+1) \end{aligned}$$

The first notation means that the contents, indicated by the parentheses, of L are exchanged (arrow head at both ends) with the contents of the address designated by the stack pointer. The double parentheses around SP indicate that the SP register contains the address of the data. Similarly, the second notation indicates that the contents of the H register are exchanged with the data at the next address above that indicated by the stack pointer.

XTHL is similar in use to XCHG. With XTHL you can use two addresses in memory to point to another memory location. However, with XTHL you must be sure the stack pointer has the same value for the second XTHL operation, or what you get back into the HL pair will not be the same as when you started. Any time you work with the stack pointer, or any stack operation, you must be careful to keep track of what the stack is doing. As you will learn in Unit 5, most stack operations change the value of the stack pointer automatically.

### Self-Test Review

11. What instruction copies the values from HL into the program counter? PCHL
12. The SPHL instruction copies the values from the H  
L into the stack pointer.
13. What instruction is sometimes called an indirect jump? PCHL
14. What does the X stand for in XTHL? exchange
15. What does the T stand for in XTHL? Top H L
16. What does the L stand for in XTHL? L register
17. Which two registers are exchanged by XCHG? DE and ~~H~~ HL
18. The contents of which address are exchanged with the H register when XTHL is executed? SP H

## UNIT SUMMARY

1. The move, copy, and exchange instructions use three addressing modes: direct, indirect, and immediate.
2. In direct addressing, the operand is the address of the data.
3. In indirect addressing, the operand is the name of a 16-bit register that contains the address of the data.
4. In immediate addressing, the operand is the data.
5. The operand may be either a number or a label.
6. All numbers start with numerals.
7. Labels are groups of letters that stand for a number or address.
8. The direct addressing instructions are as follows:

STA	Store A to operand address
LDA	Load A from operand address
SHLD	Store HL to operand address
LHLD	Load HL from operand address

9. The indirect addressing instructions are as follows:

MOV r1,M	Move contents of address HL into r1
MOV M,r1	Move contents of r1 to address HL
LDAX B	Load A from address BC
STAX B	Store A into address BC
LDAX D	Load A from address DE
STAX D	Store A into address DE
XTHL	Exchange HL with data at address SP

10. The immediate addressing instructions are as follows:

LXI	Load pair immediate
MVI	Load register immediate

11. The following instructions operate only on the data in registers:

MOV r1,r2	Copy r2 to r1
XCHG	Exchange DE with HL
PCHL	Copy HL into PC
SPHL	Copy HL into SP

12. PCHL is sometimes called an indirect Jump.
13. For XTHL, the L data goes to address SP and the H data goes to address SP + 1.
14. Only XTHL and XCHG are exchanges, the others copy the data, destroying what was there.

## **EXPERIMENTS**

Perform Experiments 7, 8, 9, and 10.



*Unit 4*

**INTRODUCTION TO  
PROGRAMMING**



## CONTENTS

INTRODUCTION . . . . .	4-3
UNIT OBJECTIVES . . . . .	4-4
PROGRAMMING LANGUAGES . . . . .	4-5
PLANNING YOUR PROGRAM . . . . .	4-8
Flow Charts . . . . .	4-9
Constructing a Flowchart . . . . .	4-15
Coding . . . . .	4-18
CONDITIONAL AND UNCONDITIONAL JUMPING . . . . .	4-22
Condition Codes, or Flags. . . . .	4-23
Jumps . . . . .	4-24
UNIT SUMMARY . . . . .	4-27
EXPERIMENTS . . . . .	4-30

## INTRODUCTION

Regardless of how complex your computer system may seem, when it is viewed from the microprocessor there are only two things you can do. You can program it, or you can connect hardware and interface it with the outside world. Anything else is just a variation on one of these two themes. In this course you are learning how to program the microprocessor. This unit is about how to plan your programs and about the jump instructions that allow computer decisions.

As you have already learned, computers cannot "think." The decisions a computer or microprocessor makes are based on the condition flags at the time of that decision. Further, all computer decisions are yes or no. There is no room for maybe or any other alternative. Many programs seem to take a multiple choice, but in fact each decision is divided into several with only two alternatives. If a computer controlled car approached an intersection, it would have to divide the three ways so that the decisions are made as a choice of two. For example, the first choice might be to either turn, or go straight. If you go straight, no further choice is needed. If you turn, then there are only two choices: right or left. An alternative to this would be to make the choices on turning as follows:

1. Turn right (yes or no)
2. Turn left (yes or no)

As a programmer, you will decide which methods your microprocessor will use to make these decisions. The first major step in programming is planning. After you have a plan, it is relatively easy to select the instructions that will follow that plan. The primary focus of this unit is on how to plan your program.

## UNIT OBJECTIVES

When you complete this unit you will be able to:

1. Explain the differences between machine code, assembly language, interpretive language, and compiler language.
2. Draw the five basic symbols used in flow charting, and explain the purpose of each.
3. Develop flow charts that illustrate step- by-step procedures for solving simple problems.
4. Explain the purpose of conditional and unconditional jumps.
5. Using the programming model of the 8085, trace the data flow and changes in values during the execution of a jump instruction.
6. Explain the relationship between the condition flags and the conditional jump instructions for the 8085.

## PROGRAMMING LANGUAGES

All computers, no matter how simple or complex, operate from instructions. The sequence of instructions to perform a given task is called a **program**. For any computer, there may be many different sets of written instructions that can be used. These sets of instructions are called **programming languages**, and as a programmer, you may write in any of these languages.

Programming languages are referred to as either high level or low level, depending on how close they are to the opcodes that are decoded within the microprocessor. Languages that are very close to those opcodes are called **low level** languages. Languages that are not closely related to those opcodes are called **high level** languages.

Another name for the opcodes is machine code or machine language. Machine code is not normally called a computer language, because you do not usually write programs in machine code. Instead, you use some notation for machine code. In this course you have and will be learning the mnemonics that are called assembly language. For each opcode, there is at least one mnemonic. You can easily determine which opcode will be produced by each mnemonic. Assembly language is the lowest level language. It is a true programming language, because it allows instructions that are not specific machine code items. These other instructions are interpreted by the assembler program, and perform such operations as defining labels and assigning the starting address of the program.

Higher level languages combine several lower level instructions into one high level instruction. For example, instead of repeatedly moving the value to the accumulator, checking to see if the port is ready, and then transferring the value to the output port; a high level language uses a single instruction such as the word PRINT. Although languages are usually referred to as either high or low, it can be said that the higher the language, the more removed from the microprocessor it is. Except for those parts of the program that deal with specific I/O ports or reserved memory locations, high level language programs can be entered into different types of computers and executed without change. Higher level languages are generally easier to learn than low level languages, because the high level languages are closer to "natural" languages, such as english.

Before a program can be executed by the microprocessor, it must be translated to machine code. Although you can translate assembly language "by hand" with paper and pencil, higher level languages require a special program to make the conversion. There are three fundamental types of language translators: assemblers, compilers, and interpreters. Regardless of the translation process, the program you write is called **source code**. The result of the translation is called **object code**. For some translators, the object code is machine code, but for others there is an intermediate level, which although it is called object code, requires some further translation.

**Assemblers** are specifically for assembly language. Some assemblers translate the language in one pass. That means they read the assembly language program from beginning to end and compose the machine code as they go. Such assemblers are restricted in that they must encounter a label definition before it can be used as an operand. Two pass assemblers read the source code twice. The first time, all labels are evaluated and assigned a numerical value. On the second pass, the final translation is made that results in object code.

**Compilers** are quite similar to assemblers, except that most compilers produce an intermediate code, that requires further linking and translation. Sometimes assembly language is used as the intermediate code. When it is required, a linker program joins required subroutines and generates the machine code.

**Interpreters**, on the other hand, not only produce executable machine code, but they also execute that code immediately. This is very useful when you are learning a language, because you get immediate feedback on the progress of your program. Like most good things, this has disadvantages, too. The primary disadvantage is execution time. Because the code is translated as you go, it takes time. In contrast, assembly language programs are usually extremely fast, because the code is tailored to the specific task. Compiled programs generally fall somewhere in-between, because they use standard routines that may not be as compact as those written in assembly language for the specific application.

## Self-Test Review

1. A set of written instructions used to program a computer is called a program LANGUAGES.
2. Written instructions that have a close correspondence to the microprocessor's opcodes are called low level.
3. Written instructions that do not closely correspond to the microprocessor's opcodes are called High level.
4. The program instructions you write are called machine code.
5. The translator produces source code.
6. Most Compilers produce an intermediate code.
7. When you use a/an Interpreters, your program is executed as soon as it is translated into machine code.
8. Generally, the fastest executing code is produced from Assemblers language.
9. The slowest execution is through a/an Compiled.

## PLANNING YOUR PROGRAM

Computer programming is done in three main phases; planning, coding, and debugging. During the planning phase, you decide what you want your program to do. In the coding phase you put those plans into the language you have selected. The debugging phase includes running the "finished" program and coding it again until it is working correctly.

Although it is quite common for one person to do all three of these phases, it is not uncommon to have teams working on a program. One person can do the planning and another can do the coding. Debugging is then done by those two working together, or possibly by a third party.

The most important part of computer programming is planning. Proper planning can save hours of coding and debugging. Although your plans will not specify which instructions will be used, you should give some consideration to which language you want to use. The lower the language level, the more detailed your plans should be.

If someone else is going to do the coding, your plans must be specific enough so that there is no misunderstanding about the desired results. But, even if you do your own coding, it is important to plan well. You cannot depend on your memory, because there may be some time between when you make your plans and when you write the code. Proper plans will leave no doubt about what you wanted to have the computer do.

Proper plans also prevent the "little bit better" syndrome that commonly plagues commercial programmers. Often, programmers get involved in refinements and extra features that unnecessarily add time to program development costs. A good plan will let you know when you are finished, because you will have done everything that is in the plan. If you think about that last statement, you will realize that it encompasses all that your plan should be; i.e. a plan is a definition of the work to be accomplished.

The details of your plan depend on the type of program you are writing. A system program would include the specific ports and addresses reserved for system use. A data base program, on the other hand, would clearly define the parameters of the data you expect to work with. It is important to anticipate as much as you can, and make notes on the as-

sumptions you have made. For example, if you are writing a data base program, be sure to identify the amount of space you want to reserve for each item. In some cases this is easy, because the length is determined by tradition. ZIP codes and phone numbers have standard lengths. Names, however, may contain only a few characters or many.

The notes you make and the hard copy of your plans will answer your questions as you proceed. You may also have to add to your plan or make it more specific. Keep your notes. You will need them when you debug and also later if you need to modify your program.

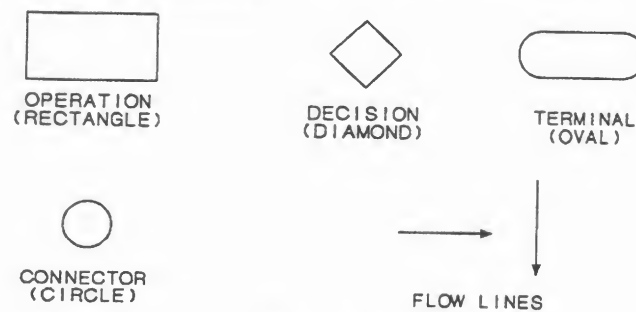
## Flow Charts

A flow chart is a diagram that identifies the sequence of events in a program. Like the schematic diagram of an electric circuit, a program's flowchart allows you to quickly identify the elements in the program. Unlike the schematic diagram, a flow chart might not be a one-to-one representation of the elements that make up the program. For example, if you are using a high level language, the boxes on the flowchart might represent each statement in the program. But if you were to look at the intermediate code, each flow chart box would correspond to several instructions.

The level of the flow chart should relate to the programming language you expect to use. If you are working in assembly language, it might help to do a high level flow chart, and then use it to construct a low level flow chart. This would mean converting each box on the high level chart to several on the low level chart.



The five most common flow chart symbols are shown in Figure 4-1. For discussion purposes, these are referred to as the oval, the diamond, the circle, the rectangle, and the flow lines. Program flow is always in the direction of the arrows -- never against the arrow. For all of the open symbols, a description of the operation is written within the symbol. Because the lines only represent flow, no symbols are required. Notice that these are only the five most common flowchart symbols. Many people use special symbols for display output, printer output, and mass storage. Although these other symbols are generally recognizable, they are not standard for all programmers.



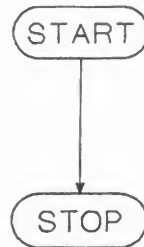
**Figure 4-1**

The Five most common Flowchart Symbols.

The oval is the symbol for a terminal point in the program. This is principally used to mark the beginning and the end of the flowchart. For most programs, there is only one beginning and one end. However, you may find it convenient to mark more than one end point. When you write the code, you should have only one exit from the program, regardless of number of ends on the flowchart. To further identify the oval as start or stop, you should write the word start or stop inside the oval.

When preparing the flowcharts for assembly language routines, you may find that more than one entry point allows you to use the common parts of the routine for several functions. For example, a routine to display a hexadecimal value, might contain a routine to display a decimal value.

Figure 4-2 shows how the terminal ovals appear in a flowchart. Notice the arrow between them. All flowcharts must have a beginning (Start) and an end (Stop). Therefore, this is simplest flowchart possible.

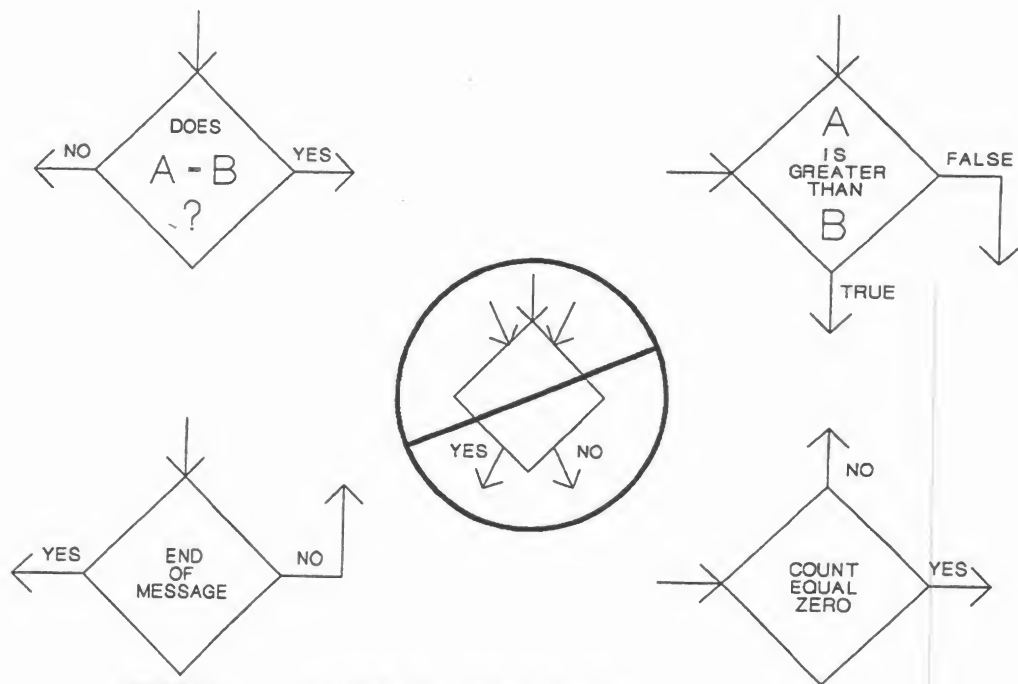


**Figure 4-2**  
The Terminal Oval.

The diamond symbol is used whenever a decision is needed. Remember, every decision box will have two, and only two arrows from it. If you think there should only be one, it is not a decision. If you think there should be more than two, you are ignoring the limits of digital computers, and you should reread the introduction of this unit. Because a program may have multiple paths, you may have two arrows into the diamond, but you must be sure they are properly marked. In the last section of this unit you will learn about the decisions that the 8085 can make.

Inside the diamond, you identify what decision is to be made. You must also identify the output lines to show which alternative they represent. Be sure that the labels on the output lines correspond to the question written inside the diamond. For example, although it is considered poor form, your diamond could say "Which is greater, A or B." With such a decision, you cannot label the outputs "yes" and "no" but you could use "A" and "B." Because the microprocessor makes comparisons in this form, it is better to label your diamond "Is A greater than B."

Figure 4-3 shows four variations on the decision symbol. By custom, the input is to the top or to the side, but seldom to the bottom. There is usually only one input, but two are acceptable. For more than two, the junction is made prior to entering the diamond. Also, the inputs and outputs are at the corners of the diamond, so the symbol in the center is not acceptable. Although true/false or other choices are permitted, the standard is yes/no.



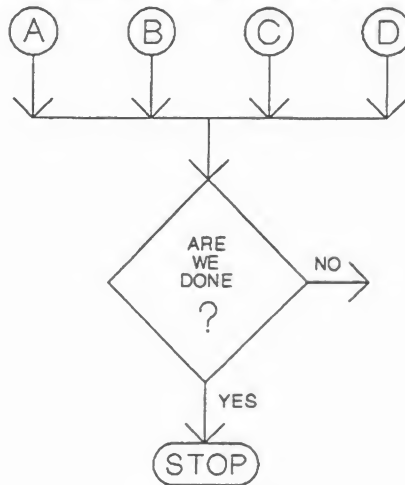
	<u>SYMBOL</u>	<u>EXAMPLE</u>	<u>DESCRIPTION</u>
1.	=	$A = B$	A and B have the same value.
2.	<	$A < B$	A is less than B.
3.	>	$A > B$	A is greater than B.
4.	$\leq$ or $\leq$	$A \leq B$	A is less than or equal to B.
5.	$\geq$ or $\geq$	$A \geq B$	A is Greater than or equal to B.
6.	$\neq$ or $\neq$	$A \neq B$	A and B do NOT have the same value.

**Figure 4-3**

The Decision Diamond and Mathematical Relationships.

To make it easier to note the decisions, certain mathematical relationships are used within the decision box. As shown, items 4, 5, and 6 have alternate symbols. Greater than or equal ( $\geq$ ) is sometimes represented by a greater than symbol over a line ( $\geq$ ). Less than or equal ( $\leq$ ) is sometimes represented by a less than symbol over a line ( $\leq$ ). Not equal ( $\neq$ ) is sometimes represented by an equal sign with a diagonal line through it ( $\neq$ ).

The circle is used to mark connection points. In many flowcharts, the structure is too complex for a single sheet of paper, or the crossing of lines might lead to confusion. If the connection is to the same page or the number of pages is few, a single letter or number identifies the connection. Two circles containing the same letter or number indicates the connection. To avoid confusion, you should make connections only to one other point. In other words, only two circles will have the same number. If you have flow from several points entering a single point, you would have several connector circles feeding together as shown in Figure 4-4.



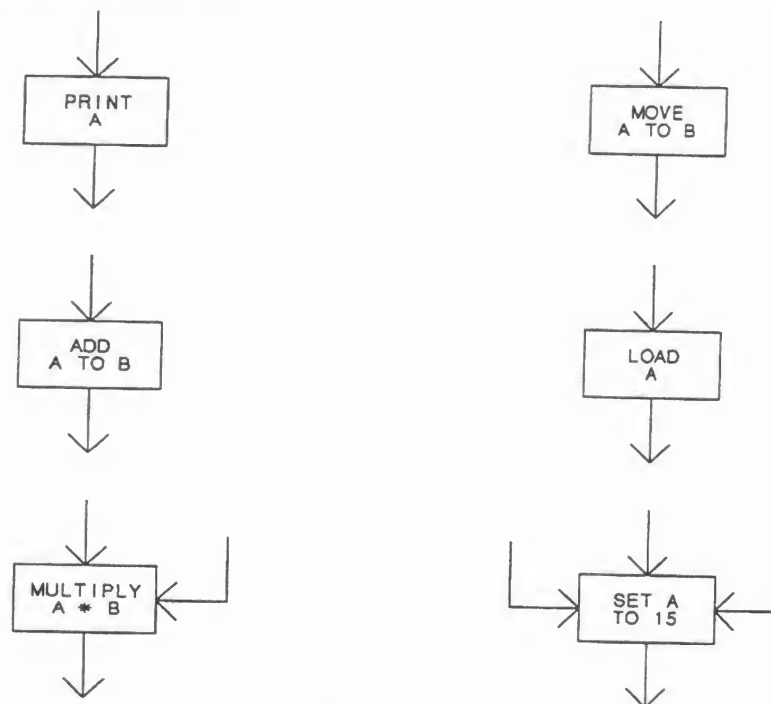
**Figure 4-4**  
The Connection Circle.

When flowcharts require many sheets, some method must be used to make it easier to determine the location of the other connector circle. Two of the most commonly used are the page number method and the index method.

Under the page number method, the circle contains a letter and the page number of the corresponding connection. For example, on page 5 you might put a connection to 23A would mean page 23 circle A. On page 23 you would then have a circle with the designation 5A, indicating that the corresponding connection is circle A on page 5.

Under the index method, all connector circles are identified in pairs, and a record of the locations is placed in an index. The index simply lists all connections, and the pages on which they appear. For example, if the pair of circles marked A appear on pages 1 and 14, the index would have a listing of "A 1,14" to show those locations.

The rectangle symbol is used for all arithmetic, logic, input, output, load, store, and other operations. In short, anything that is not a terminal, a decision, or a connection must be an operation. Figure 4-5 shows some examples of operations. Notice that an operation can have only one output line. More than one output would require a decision. However, it is permissible to have multiple inputs. As mentioned earlier, depending on the level of the flowchart, some operations may require several opcodes. For example, in Figure 4-5 the operations PRINT A and MULTIPLY A\*B require several opcodes.



**Figure 4-5**  
The Operations Rectangle.

For frequently repeated operations, such as print and multiply, you can prepare separate low level flowcharts. Then, when you code your program, refer to that flowchart routine to prepare the code. As you will learn in Unit 5, it is easy to code the routine once, and then, use it whenever it is needed.

The flow lines are used to connect the other flowchart symbols. You must use an arrowhead where a line enters a box—a line without an arrowhead is an output from the symbol. You may use arrowheads at other points to clarify the sequence of events. Some confusion may result if two flow lines cross but are not connected. In general, a connection is assumed unless it is obvious that the lines are not connected.

## Constructing a Flowchart

You saw an elementary flowchart when you learned about the terminal symbol. All that remains is to insert the various boxes that define your program. Obviously, this is a gross simplification of the situation, and what you really need is a plan of action. To prepare your flowchart, first make a list of all you want your program to do, and the order you want it done. As an example, Figure 4-6 lists the items to be done by a simple calculator program. This list will form a basis for the program, but it is much too simple to be of any real value to us yet. Also, this may not be the order in which the steps should be performed, and it does not indicate if any steps should be repeated.

1.     Input the first value.
2.     Input the operation.
3.     Input the second value.
4.     Perform the operation.
5.     Display the results.

**Figure 4-6**

A List of Program Functions.

Before you draw your flowchart from this information, you must elaborate, or expand it. For example, it is not enough to say "input the operation;" you must specify what operations you are going to allow. Other program considerations, such as allowable values, do not need to be considered now, unless they affect the procedure. Therefore, it is not necessary to know how large a value is allowed, but it is necessary to know what number base you are using.

These changes bring us to the expanded list of program functions in Figure 4-7. Notice the following changes. First, the number base is requested. Second, The operation is requested after the second value. (This avoids the need for an additional key press to perform the operation.) Third and last, all possible operations are listed.

1. Establish number base (Binary, Decimal, Hexadecimal)
2. Input the first value.
3. Input the second value.
4. Input the operation.
  - Add
  - Subtract
  - Multiply
  - Divide
5. Perform the operation.
6. Display the results

**Figure 4-7**  
Elaborated List of Program Functions.

Now you can build a flow chart, but you will have to make some interpretations as you do. For example, the first step says to establish the number base (binary, decimal, or hexadecimal). Because the flowchart allows only two way decisions, the base selection will require at least two decision boxes. You might think it would take three, but you can assume that if it is not the first or second choice, it must be the third.

Similarly, the four arithmetic operations can be done by three decisions with the assumption of the fourth. Based on this, and including a test to see if you are through with the calculator, you arrive at the flowchart shown in Figure 4-8A and 4-8B. The primary reason for showing this in two pieces is to demonstrate the connector symbols.

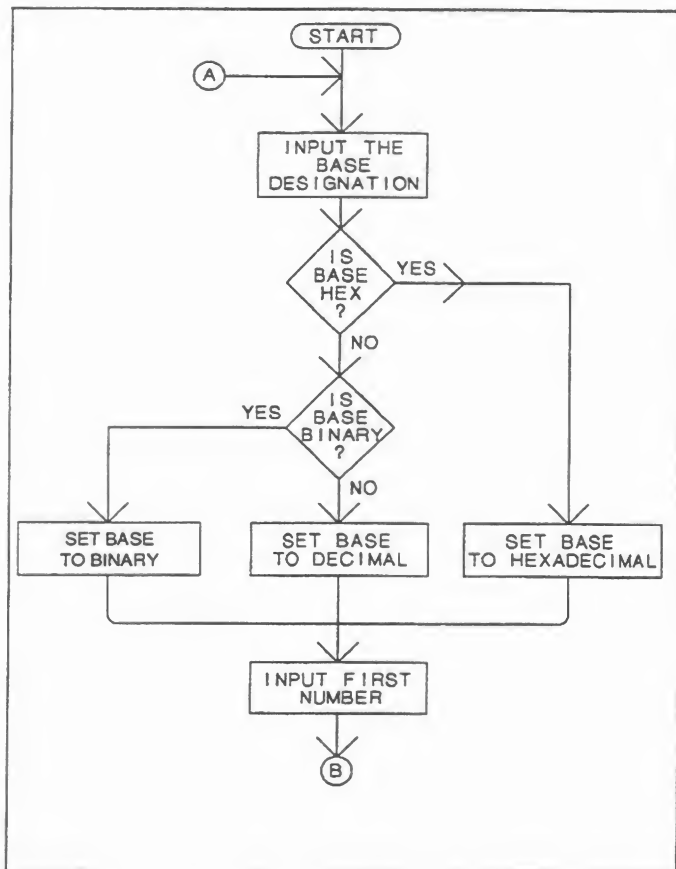


Figure 4-8A

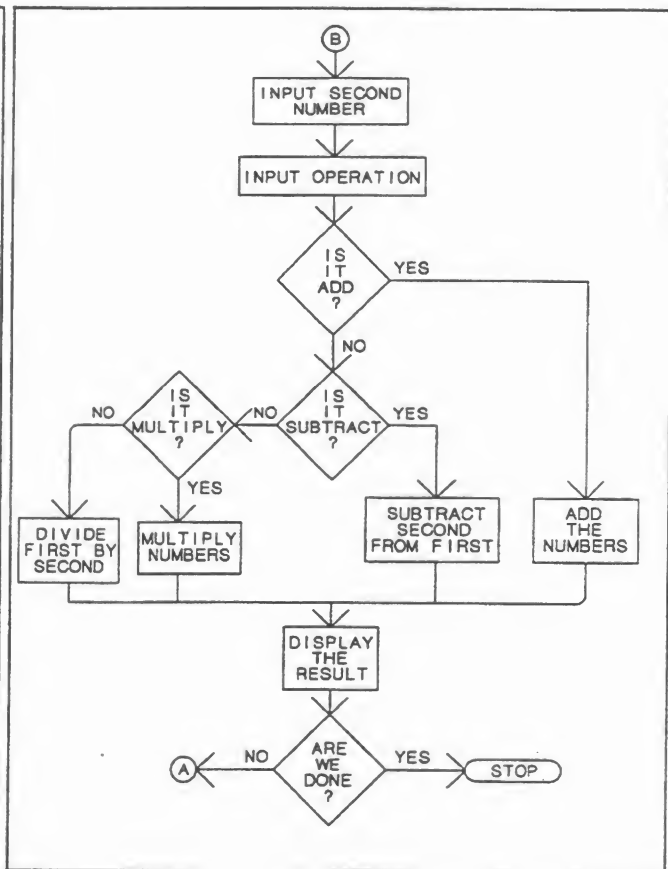


Figure 4-8B

Example Flowchart.



## Coding

Once you have completed your plans and flowchart, you are ready to code the program. This is not to say that the plans are absolute, concrete, and unchangeable. However, you must remember that changes from the planned program usually result in a decrease in quality or an increase in development time. Another aspect of the problems related to deviations from the plan occurs if the deviations occur during the coding. The primary symptom of this problem is that the programmer gets so involved in the added feature, that the more important functions are not given proper treatment.

The language you are using, and your familiarity with it, will determine whether you can write your code directly from the flowchart. You may find it helpful, or even necessary, to prepare coding flowcharts for individual operations in the overall flowchart. For example, the step that says "input first number" necessarily requires the input of the individual characters and a test to see if the entry is complete. But if you are working with assembly language, you may also need to include the tests to see if a character has been received from the keyboard. As you can see, the difference between the flowchart in Figure 4-8 and the code for the program can be considerable.

Another related factor is the interface with the specific computer you are using. All computers have several levels of software operation. For discussion purposes, we will call them monitor, BIOS, and kernel. The monitor is the program in ROM that gets the computer going and provides some basic functions. The BIOS, basic input/output system, contains the routines for communication with all the standard peripherals. And, the kernel assigns standard names or labels that associate the peripherals with the operating system (usually disk based). Properly written programs make full use of the next level of operation. Therefore, if your code is written to operate with the operating system, your input and output with standard peripherals will be through kernel labels and names that identify the entry points for those routines. If, on the other hand, you are writing kernel software, you will provide a link to the BIOS.

Unless you are writing code for the lowest levels of control (monitor or BIOS), you must NOT communicate directly with the "usual" hardware. Kernel level software allows you to prepare programs that will work with successively newer generations of a particular machine. For example, the operation of the 8085 is upward compatible with the operation of the 8088. Similarly, the operation of the 8088 is upward compatible with the 80386 (and there are several levels of compatibility in between). Therefore, by writing your 8085 programs to work through the kernel for a particular operating system, your programs will also work on 8088 versions of the operating system, and on 80386 versions of the operating system. All this works, even though the specific hardware for the newer machines is different from the older ones.

So, what does all this mean to you as you prepare programs for the 8085 trainer? The answer is that you should make yourself familiar with the I/O routines and whenever possible, or not specified otherwise, you should write your programs to make use of those routines. When you want to send a character to the display, you should send it to the display subroutine. You should not write a routine that directly addresses the display.

These are all philosophical aspects of coding. They are very important, because they help you decide what routines to use, and when you should use an existing routine or write your own.

The next aspect of coding is selecting the operations you will use. There are many ways of doing any particular task, and your selection of code will depend on your need for computation speed or code clarity. These two are often mutually exclusive, that is, you cannot have both at the same time. For example; the fastest executing code may rely on some clever bit manipulation that is difficult to understand by looking at the code. The more obvious methods of moving and combining data may require substantially more time to execute. Only a lot of programming experience and some trial and error coding will determine the best code for a particular situation. A good programmer needs to continually study to see what others have done and regularly up-grade his technique.

Two things that will help you are to make notes on where you are storing data and to keep in mind where data must be for use by the I/O subroutines. If, for example, a character must be in the accumulator before it can be sent to the display subroutine, it makes sense to move it from memory directly to the accumulator, rather than through another

register. As another example, let's say you write a routine to copy data from one place in memory to another. The number of bytes is indicated by B-C, the source address by D-E, and the destination by H-L. It is much easier to write that information down now, than to figure it out when needed later.

To make it easier to remember what is happening, you will want to use the same registers each time you perform an operation. In many cases the use of the register is dictated by the instruction set, but in those cases where you have a choice, keep the choice the same so you can remember what you have done. However, this is not a substitute for keeping notes.

Perhaps the most important part of coding is to make comments in the code or on paper to tell you what you have done. Although you may know something perfectly well when you first generate the code, you may forget it if much has happened since you last looked at the code. This is particularly true of acronyms used as labels. Be sure to make a note of what that acronym stands for, and also make a note of what that really means. It is not wrong to identify a variable as SD1. To note that it means side one may seem trivial but in six months when you look at the code again, you might forget and think that it means system drive 1. You will also forget whether side one is the left side or the right side, the top or the bottom. Further, unless you have made good notes, you may even forget that the routine solved a tetrahedron instead of a triangle. Comments are not just nice, they are vital if you ever want to look at the source code again. Imagine that you have wired your car and used different colors to identify the circuits, but you did not make a list of those circuits. A year from now, when the circuit fails, you won't even know which color wire you need to follow.

## Self-Test Review

10. The three phases of computer programming are phases, planning, and coding.
11. The flow chart symbol for a terminal point is the arrows.
12. The flow chart symbol for a decision is the Diamond.
13. The flow chart symbol for a/an operation is the rectangle.
14. The flow chart symbol for a/an connector point is the circle.
15. The expression  $A \neq B$  in a decision box means A ( $\neq$ ) B.

## CONDITIONAL AND UNCONDITIONAL JUMPING

In the previous section you learned about decision blocks and saw a flow-chart for a program that looped back to the beginning if the operator desired. These operations are made possible by conditional and unconditional jumps, or branches.

As you might guess from these names, conditional jumps correspond to the diamond, or decision, boxes on the flow chart, and unconditional jumps may be represented either by circles or simply by a line. The opcode for an unconditional jump, JMP, is followed by the address, or destination, of the jump. In assembly language that destination is usually identified by a label. Therefore the mnemonics JMP BEGIN and JMP 8000H would mean the same thing if BEGIN was the label for address 8000<sub>16</sub>.

The conditional jumps are listed in Figure 4-9. Notice that the first four jump when the specified flag is in a zero condition and the last four jump instructions jump when the specified flag is set to 1.

<u>Mnemonic</u>	<u>Meaning</u>	<u>Flags Checked</u>
JNZ	Jump Not Zero	Jump if Z=0
JNC	Jump No Carry	Jump if C=0
JPO	Jump Parity Odd	Jump if P=0
JP	Jump Positive	Jump if S=0
JZ	Jump Zero	Jump if Z=1
JC	Jump Carry	Jump if C=1
JPE	Jump Parity Even	Jump if P=1
JM	Jump Minus	Jump if S=1

**Figure 4-9**  
Conditional Jumps.

Like the unconditional jump, the opcode for each of these is followed by the two-byte destination address. When the jump is executed, the next two bytes are put into the instruction pointer. Consequently, the next instruction to be performed will be the opcode at the designated address. For a conditional jump, the specified flag is tested before the value is copied to the instruction pointer.

## Condition Codes, or Flags.

As you will remember, the flags (condition codes) are set or cleared by the last arithmetic or logic operation. At this time it becomes more important what the flags are and what their value means.

\* The zero flag is set to 1 only if the arithmetic operation resulted in a zero value. For example, when you subtract ten from ten the result is zero. What is not as obvious is that the result in the accumulator may also be zero if the result of the last operation has a value of  $256_{10}$ . For example, if all bits in the accumulator are 1's (a value of  $255_{10}$  or  $-1_{10}$ ) and you perform an INR A instruction, the result is zero. Thus, the zero flag would be set.

The carry flag is set when an overflow or borrow is generated by an arithmetic operation. Naturally, such a condition might require special consideration. If you are performing a complex math calculation and a borrow occurs, you may want to adjust the answer to reflect that condition. As an alternative, you might also want to change the notation and recalculate so that the value is within range. Either of these would correspond to a decision on the flowchart.

\* The carry flag can also be controlled by two special opcodes. These are STC ( $37_{16}$ ) and CMC ( $3F_{16}$ ). STC is very straight forward, it stands for "set the carry" and that is exactly what it does. After an STC opcode is executed, the carry flag will be set to 1. There will be no other changes. The CMC opcode toggles the carry flag. Therefore, whatever the carry flag was before (and you can determine that with a JC or JNC instruction), after CMC the value in the carry will be changed. If it was a one, it will be a zero. And, if it was a zero before, it will be changed to a one. Because these opcodes affect only the carry flag, you can use them wherever you want to use them. Further, these opcodes are unique, because the carry flag is the only flag that has special instructions to control it.

You might question the value of this, but as you will remember, the rotation instructions include the carry, so this bit can appear to be a part of the accumulator. As such, it is nice to be able to control that single bit.

\* The parity flag indicates whether the number of bits that are set (1's) is even or odd. If the number of 1's is even (0, 2, 4, 6, or 8) then the parity flag is set. When the number of 1's is odd (1, 3, 5, or 7), the parity flag is cleared to 0.

The sign flag is used to indicate that the value of the last operation is negative. What it actually indicates is whether the high bit of the last operation is set or cleared. As you have already learned, the eight bits of a byte can represent any of 256 values. Therefore, if the normal values are numbers from 0 to 255<sub>10</sub>, the sign flag may not mean the value is negative, unless the last operation was a subtraction. If you add 127<sub>10</sub> and 1, the result will be 128<sub>10</sub>. Because the high bit of value 128<sub>10</sub> is set, the sign flag will be a 1. In writing your program, you must be aware of the possible values of the numbers before and after the calculation, so you can interpret the meaning of the flags. When the sign flag is set it might mean the value is minus, but it might also mean the value is greater than 127<sub>10</sub>. You must code the program so that the subsequent operations are correct in accordance with the meaning of the value.

## Jumps

Without jumps your programs could only have a single path. Further, it could only execute once and you would have to start it again from the beginning. Programs without jumps are called linear programs, and although there is nothing inherently wrong with linear programs they are, by nature, very simple.

\* By using conditional jumps, you can write **branched** programs. One special type of branch is called a **loop**. A loop is a program or part of a program that is repeated. Although you can form a loop with an unconditional jump, the loop would be continuous, or endless. Endless loops are usually a problem that must be avoided.

The only way to stop an endless loop is by turning off the computer. One situation where an endless loop is used is for the monitor program in a dedicated computer system. Another situation is in troubleshooting, where you want the same addressing and data patterns to repeat continuously. In this latter situation, many technicians will include a keyboard check and conditional exit, so that they do not have to turn off the computer and reboot to continue.



As you have learned, there are four flags you can test to control the conditional jumps. Your selection of whether the flag is a 1 or a 0 is somewhat arbitrary. The decision is usually based on what provides the fewest jumps. for example if part of the program is executed only under certain conditions, you would jump around that routine except when the condition is met. Figure 4-10 is a listing of such a program segment.

```

MOV      A,M      ;Put value into A
CPI      07BH     ;Compare to "{" character
JP       NOCHG    ;Jump if positive (no change)
CPI      061H     ;Compare to "a" character
JM       NOCHG    ;Jump if minus (no change)
ANI      0BFH     ;Make upper case (clear bit 6)
NOCHG    MOV      M,A      ;Put revised value back

```

**Figure 4-10**

Program with a Conditional Jump.

In this sample, the sign flag was established by comparing the value (in the accumulator) first with 7B<sub>16</sub> and then with 61<sub>16</sub>. If the value is greater than 7A<sub>16</sub>, which is a "z", the first jump to NOCHG will be taken and no change will be made. If the value is less than 61<sub>16</sub>, which is "a", the second jump will be made. Only values between 61<sub>16</sub> and 7A<sub>16</sub>, ("a" through "z") will be masked by the ANI 0BFH instruction. This will convert those lowercase letters to their corresponding uppercase value.

One of the most important things to consider is the value used for comparison. In this example the first value, 7B<sub>16</sub>, is one more than the highest allowed value. This is because a zero result is considered positive. If the value in the accumulator is 7B<sub>16</sub> or greater, the remainder when 7B<sub>16</sub> is subtracted (same as compared) will be zero or more. In contrast, when the lower value is compared, the jump is made if the result is minus. Therefore, a zero should not cause a jump and the value for "a" must result in a zero. So for the lower end, the comparison is with the ASCII value for the "a" character. This is important, since if you change your test from set to cleared, you may also have to modify the operation before it.

This is not the only way this conversion could be made. Another way you could do it is to test the carry flag to see if a borrow was required by the CPI instructions.



In writing programs, you must keep your mind open to the other ways things can be done. If your program runs too slow, you will probably make the greatest improvement, by changing your approach. Another thing you can do, is use the same comparison for more than one conditional jump. It is important to keep track of what operations affect the flags and which ones do not.

### Self-Test Review

16. JPO means to jump if the jump flag is 1.
17. The two jump instructions that test the carry flag are Jp and Jm.
18. The parity flag is set (1) whenever the number of bits set (1) by the last arithmetic operation is even? 4 6 8 even.
19. Programs that have no jumps are called loop programs.
20. Each 8085 jump opcode is followed by 2 byte(s) which are put into the instruction pointer if the jump is to be taken.

## UNIT SUMMARY

1. The sequence of instructions to perform a given task is called a **program**.
2. The different sets of written instructions that can be used to program a computer are called programming **languages**.
3. Languages that are very close to the opcodes are called **low level** languages.
4. Languages that are not closely related to the opcodes are called **high level** languages.
5. Another name for the opcodes is **machine code** or machine language.
6. **Assembly** language is the lowest level language.
7. Higher level languages are generally easier to learn than low level languages.
8. Before a program can be executed by the microprocessor, it must be translated to machine code.
9. There are three fundamental types of language translators: **assemblers, compilers, and interpreters**.
10. The program you write is called **source code**. The result of the translation is called **object code**.
11. **Assemblers** are specifically for assembly language.
12. **Compilers** normally produce an intermediate code, that requires further linking and translation.
13. **Interpreters** not only produce executable machine code but also execute that code immediately.
14. Computer programming is done in three main phases; **planning, coding, and debugging**.
15. During the **planning** phase, you decide what you want your program to do.
16. In the **coding** phase you convert the plans into the language you have selected.

17. During the **debugging** phase you run the "finished" program and recode it until it is working correctly.
18. A **flowchart** is a diagram that identifies the sequence of events in a program.
19. The five most common flow chart symbols are the **oval**, the **diamond**, the **circle**, the **rectangle**, and the **flow lines**.
20. The **oval** is the symbol for a terminal point in the program, which are the beginning and the end.
21. The **diamond** indicates a decision.
22. The **circle** indicates a connection to another point on the flowchart.
23. The **rectangle** is used for all arithmetic, logic, input, output, load, store, and other operations.
24. The **flow lines** connect the other flowchart symbols.
25. Coding is done after the plans, including the flowchart, are completed.
26. Deviations from the plans usually result in greater development time or reduced program capabilities.
27. Your program should work with and use the next lower level of computer control.
28. The 8085 jump instructions are as follows:

<u>Mnemonic Meaning</u>		<u>Flags Checked</u>
JMP	Jump always	None
JNZ	Jump Not Zero	Jump if Z=0
JNC	Jump No Carry	Jump if C=0
JPO	Jump Parity Odd	Jump if P=0
JP	Jump Positive	Jump if S=0
JZ	Jump Zero	Jump if Z=1
JC	Jump Carry	Jump if C=1
JPE	Jump Parity Even	Jump if P=1
JM	Jump Minus	Jump if S=1

- 29. The **zero** flag is set to 1 if the last arithmetic operation resulted in all zeros in the register.
- 30. **STC** sets the carry flag and **CMC** compliments it.
- 31. The **carry** flag is set to 1 when an overflow or borrow is generated by an arithmetic operation.
- 32. The **parity** flag is set to 1 when the number of 1's in the register after the arithmetic operation is even.
- 33. The **sign** flag is set to 1 when the value of the last operation is negative.
- 34. The use of conditional jumps results in a **branched** program.
- 35. A **loop** is a program or part of a program that is repeated.

## **EXPERIMENTS**

Perform Experiments 11, 12, and 13. After you finish the experiments, return to this unit and complete the Unit Examination.

*Unit 5*

**STACK OPERATIONS AND  
SUBROUTINES**

## CONTENTS

CONTENTS .....	5-2
INTRODUCTION .....	5-3
UNIT OBJECTIVES .....	5-4
WHAT IS A STACK .....	5-5
The 8085 Stack .....	5-6
AUTOMATIC STACK ACTIVITY AND SUBROUTINES . . .	5-10
INSTRUCTIONS THAT CHANGE THE STACK .....	5-16
UNIT SUMMARY .....	5-23
EXPERIMENTS .....	5-26

## INTRODUCTION

Now you are familiar with jump instructions. You might imagine that it would be nice to not only jump to another part of the program, but to come back as soon as that routine is done. This capability was thought of very early in computer development, and all modern microprocessors have special instructions and facilities to handle this. One significant facility is the stack.

The stack was invented to store values during temporary excursions from the main program. When you "call" a routine, the stack holds the return address. When you "return" from the routine, that address is taken from the stack and put into the program counter. The stack is also a convenient place to save values, because they can be quickly restored.

There are several different kinds of stacks, and this unit will show you the two most common. Although the 8085 uses only one kind in association with its stack pointer, you should know how the other kind works.



## UNIT OBJECTIVES

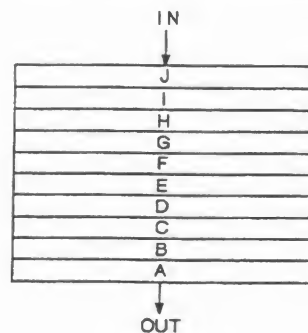
When you complete this unit you will be able to:

- 1. Define the terms call and return, and describe their affect on the stack.
2. Write simple programs using the stack to store and retrieve data.
3. Explain the operations performed by the following instructions: CALL, CZ, CNZ, CC, CNC, CPO, CPE, CP, CM, RET, RZ, RNZ, RC, RNC, RPO, RPE, RP, RM, PUSH, and POP.
4. Identify the two principal types of stacks.
5. Identify the standard type of stack used by the 8085.
6. Explain the operation of the stack and stack pointer as they relate to XTHL, SPHL, LXI SP, DCX SP, INX SP, and the RST instructions.
7. Explain the use of assembler directives and identify ORG, CSEG, DSEG, EQU, DB, DW, and DS.

## WHAT IS A STACK

Before you learn how to work with the stack, it is important to understand what a stack is and what kinds there are. In a computer, a stack is a series of temporary storage locations that permit easy input and removal of information. A stack can be able to hold only a single item or several hundred. One characteristic of stacks is that the items are in consecutive locations. It is quite unusual and generally poor practice to remove, add, or change the items in the center of the stack. Finally, there are two basic kinds of stacks. They are identified by the way they handle data. They are called FIFO and LIFO.

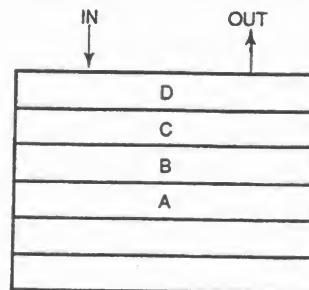
A FIFO stack performs operations on a first in, first out (FIFO) basis. This type of operation is also called a **queue**. (This is pronounced the same as the letter Q.) The important point is that items are taken out of the stack in the same order they are inserted. Figure 5-1 demonstrates this. The information was input in alphabetical order, with the A first. When data is taken from the stack, the A must come out first, followed by the B, C, and so forth, until item J is removed.



**Figure 5-1**  
The Queue or FIFO Stack.

Queues are used for instruction processing and other applications, where the first item must remain first. One advantage of a queue is that it is easy to determine when it is full. The disadvantage is that you usually have to "read" through the earlier entries to get to the latest.

LIFO (Last In, First Out) stacks output the last item first. LIFO stacks are most useful when the last item in is the first item wanted. Figure 52 illustrates a LIFO stack. Again, item A was the first item placed on the stack. This time however the items are removed in reverse order and item D must come off the stack first. This type of stack is sometimes called a “push up” stack because it resembles plates on a spring loaded holder. As each plate is put on the stack, the spring yields and the stack goes down. When the top plate is removed, the spring pushes the next plate up to the same position.



**Figure 5-2**  
The LIFO Stack.

## The 8085 Stack

The standard stack operation for the 8085 is LIFO (last in, first out). However, the stack builds toward the low memory address, so it is frequently imagined as a downward stack. This is because the lowest memory address is called the “bottom” of memory and the highest address is called the “top” of memory.

The current stack address is determined by the **stack pointer** register in the CPU. If the stack pointer is at address  $6FFF_{16}$  and two items are put on the stack, the first will go to address  $6FFE_{16}$ , and the second will go to  $6FFD_{16}$ , which is one address lower. Removing items from the stack is done in reverse order. In the previous case, the first item removed will be from address  $6FFD_{16}$ . The second item removed will come from address  $6FFE_{16}$ .

Because the 8085 stack is part of memory, and must be both written and read, it must be somewhere in RAM. However, many locations in RAM are reserved for specific purposes. For example, the ETS-8085 system uses RAM below address  $6840_{16}$ .

One of the most important things in working with a stack is that you must initialize the stack and/or know its current status. It is customary when working within an operating system for the user to establish and use a separate stack from the system. For example, if you know the area from  $6840_{16}$  to  $6FFF_{16}$  is not used by the system, you might assign a stack pointer value of  $6FFF_{16}$ . Your main program might then safely operate starting at address  $7000_{16}$  or higher. Sometimes the operating system specifies that you can (or must) use the system stack. But regardless of what stack you are using, you must always leave the stack and stack pointer in the same condition as when you found them.

Briefly, consider the situation when your program starts. The stack pointer is marking the system stack. Your job is to save that stack pointer value and establish your own stack pointer. Later, when the program is over you will have to restore the values.

Although there is more than one way to do this, the tricky part is getting the current value of the stack pointer. Even though you have learned some special instructions that involve the stack, such as SPHL and XTHL, none of these reads the stack pointer. Amazingly, there is only one instruction that can get this value for you. Recall that the DAD SP instruction adds the stack pointer to the HL pair. Therefore, if you zero HL and DAD SP the result is that the current stack pointer value will be in the HL register. It is then a simple matter to store HL (SHLD) in a specific memory location.

With the system stack pointer safely saved, you can establish your own stack pointer and continue. Before you return control to the system, you can restore the value with LHLD and SPHL. This seems complicated, and for that reason many systems that use the 8085 include enough area for you to use the system stack. Many newer microprocessors have special instructions that establish a new stack or restore the old stack with a single instruction.

## 5-8 STACK OPERATIONS AND SUBROUTINES

---

```
BEGIN      LXI      H,0           ;Zero out H and L
           DAD      SP           ;Copy SP into HL
           SHLD     STKSTR       ;Save the stack pointer
           .
           . ;This is where the program goes
           .
           .
           LHLD     STKSTR       ;Get SP out of storage
           SPHL                     ;Restore the original stack
END                                     ;This will have the appropriate
                                     ; instruction to return to the
                                     ; operating system.
```

**Figure 5-3**

Stack Control for Program Entry and Exit.

Figure 5-3 illustrates the beginning and end of a program. In some cases you will have to precede the LXI instruction with an SHLD to save the old HL value. The specific code that returns you to the operating system will be different depending on the level you are working with. It will probably be an RST (You will learn about these in Unit 6.) instruction, but it may be different under the disk system than at the monitor level. Further, if your computer works under different disk operating systems, each might have a different return function. In some systems you just jump to address 0000<sub>16</sub>.

### Self-Test Review

1. FIFO stands for \_\_\_\_\_, \_\_\_\_\_.
2. LIFO stands for \_\_\_\_\_, \_\_\_\_\_.
3. Another term for a FIFO stack is a \_\_\_\_\_.
4. What is the only 8085 instruction that can get the stack pointer into another register? \_\_\_\_\_
5. It is generally poor practice to delete, add, or change items in what part of the stack? \_\_\_\_\_

## ^AUTOMATIC STACK ACTIVITY AND SUBROUTINES

There are many 8085 instructions that automatically perform certain stack operations. These are calls and returns. The call and return instructions are used to construct parts of the program called **subroutines**. A subroutine is a program segment that can be performed at any time when required by another part of the program. Subroutines perform specific functions, such as displaying something or determining some mathematical value.

The stack is the key to allowing these subroutines to work. When the 8085 encounters a call instruction, two things happen. First, the address of the next instruction is placed on the stack. This is easily determined from the value currently in the program counter. Second, the program counter receives the argument, or destination, of the call.

At the end of the called routine, when you want to return to the place you left, you can simply return to the address at the top of the stack. To allow this, the 8085 instruction set contains a number of return instructions. Now, let's examine the call and return elements a little closer.

Figure 5-4 Shows the address values for the stack. For this illustration, the starting stack pointer value is  $6FFF_{16}$ , and the program counter is at  $7000_{16}$ . When the CALL instruction ( $CD_{16}$ ) is executed, the address of the next instruction  $7003_{16}$  is sent to addresses  $6FFD_{16}$  and  $6FFE_{16}$  as shown. The stack pointer will be decremented by two to a value of  $6FFD_{16}$ , so that it points to a new stack address.

<u>Address</u>	<u>Content</u>	<u>Remarks</u>
6FFC	XX	
6FFD	03	New stack pointer value. Location contains
		Low byte of address 7003 on stack.
6FFE	70	High byte of address 7003 on stack.
6FFF	XX	Beginning stack pointer value--empty.
7000	CD	First byte of CALL 7007 instruction.
7001	07	Second byte of CALL 7007 instruction.
7002	70	Third byte of CALL 7007 instruction.
7003	C3	First byte of instruction after CALL.
7004	00	Second byte of instruction after CALL.
7005	70	Third byte of instruction after CALL.
7006	76	HLT instruction.
7007	C9	RET instruction.

**Figure 5-4**

Stack in Action (A Single Call).

The value at  $7007_{16}$ , in this case, is an RET (return) instruction. When a return instruction is executed, the two values starting at the stack pointer address and the next higher address are returned to the program counter (in this case, the number  $7003_{16}$ ) so that will be the next instruction executed. In this figure, the addresses are listed in order from lowest to highest. This makes it slightly difficult to read the address values from the listing. This is the primary reason the comments are listed here

In addition to the CALL and RET opcodes, the 8085 also has eight conditional calls and eight conditional returns. Like the conditional jump instructions you learned about in the last unit, each of these conditional instructions check a flag to determine whether to make the branch or not. It is important to remember that the call and return instructions are only related, not bound together. When you make a conditional call, you do not have to use a conditional return.

<u>Flag</u>	<u>Call if 0</u>	<u>Call If 1</u>	<u>Return If 0</u>	<u>Return If 1</u>
Z (Zero)	CNZ	CZ	RNZ	RZ
C (Carry)	CNC	CC	RNC	RC
P (Parity)	CPO	CPE	RPO	RPE
S (Sign)	CP	CM	RP	RM

**Figure 5-5**

Conditional Calls and Returns.

Figure 5-5 is a list of the flags and the conditional calls and returns that are related to them. Like CALL and RET, these instructions work with the stack and stack pointer registers to effectively change the course of program execution. Let's consider some of the uses for these instructions.

You can use the conditional calls to temporarily leave the main path of program execution and perform some other operation. For example, you might read the keyboard and compare it to some value. Then, if the value matches you call the subroutine. If the value does not match, you continue execution.

Similarly, the conditional return instructions allow you to either return from the subroutine or continue until another return is encountered. However, it is important to remember that you must eventually execute a return. You **MUST NOT JUMP** back to the main program from a subroutine. For every executed call, the microprocessor must execute a return. The reason for this is simple. If you do not return after a call, the stack will still have the return address. You must be sure that all items placed on the stack are removed or disposed of properly.



## 5-12 STACK OPERATIONS AND SUBROUTINES

---

Regardless of whether you are using the system stack or your own, you should restore the stack to its previous condition at the end of the subroutine. Without changing the stack pointer, this means the items at the top of the stack are again at the top of the stack. Thus, if your main program has put items on the stack, it (the main program) will be able to pick them up again regardless of what has happened in the mean time. Your subroutine may call other subroutines, which can call still other subroutines, and at the end the stack will be correct.

Another group of instructions, closely related to CALL are the RST instructions. These instructions, RST 0 through RST 7, allow you quick access to subroutines in the low end of memory. Like other calls, the RST instructions begin by putting the next address on the stack. However, RST instructions require only a one byte opcode. You do not need to specify the destination address, because the RST number (0 to 7) defines the destination. To determine the destination, multiply the RST number by eight. The result will be the subroutine addresses listed here.

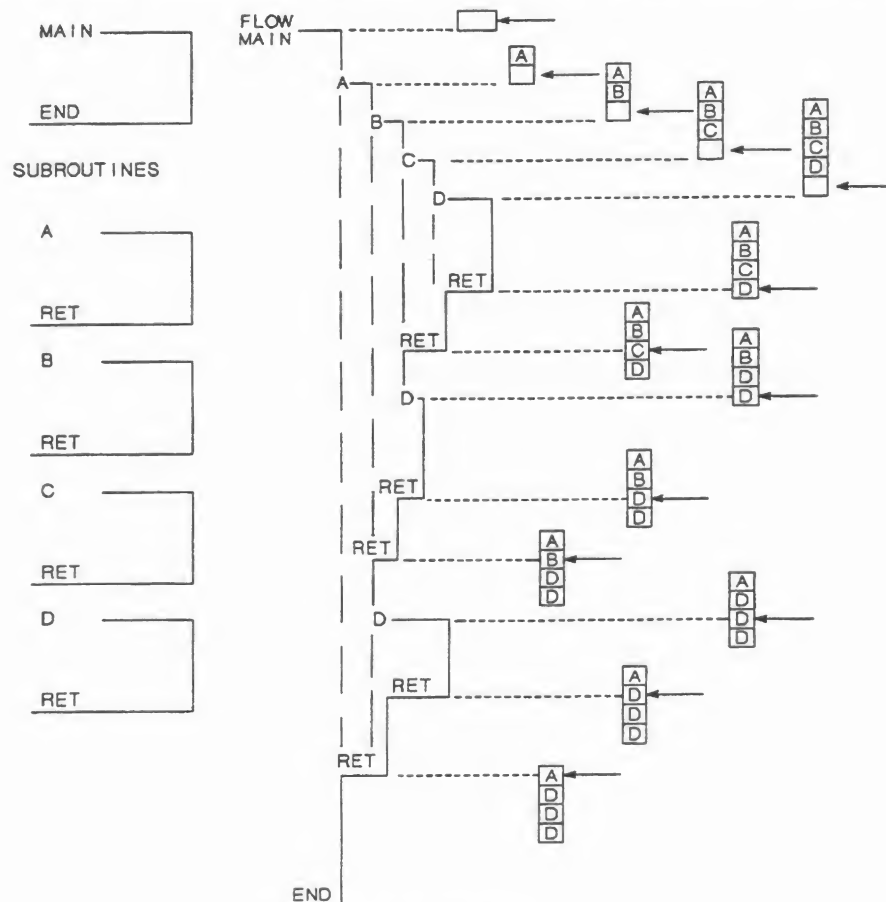
<u>RST</u>	<u>Hex Address</u>
RST0	0000
RST1	0008
RST2	0010
RST3	0018
RST4	0020
RST5	0028
RST6	0030
RST7	0038

As you have probably noticed, this leaves only eight bytes for the subroutine. To accommodate larger routines, the normal procedure is to put a JUMP at the RST address. You might think this is a waste, because the microprocessor has to change the program counter twice in short succession. This is true, but if the subroutine is used often enough, the two byte savings at each use quickly makes up for the delay in execution time. Some programmers think it is smart to determine the value at the RST point and make the subroutine call directly. This is good, until the system is upgraded and the value at the RST address is changed. Then, the program no longer works. This has happened often on programs for the IBM PC. The programs work directly with the firmware or hardware,

and therefore would not run on a "compatible" system. Then, both the user and the programmer get upset. You can avoid this frustration in your programs by using only the entry points specified and allowed by the next level of control.

The final point to consider with regard to subroutines and RST instructions is what happens when one subroutine calls another. This is not complicated. As you have already learned, the stack pointer is decremented to point to a new location after each call. Therefore, each call (made before the return) places its own return address on the stack. If several subroutine calls are made, the stack will contain several return addresses. This is why it is so important to know what the stack is doing. Your stack must have enough room to build without running into other data. Also, your program must return from each subroutine to the instruction after the previous call.

Figure 5-6 should help you visualize the stack and the calls. The first column illustrates how the main program and subroutines are stored in memory. The second column traces the flow during execution. Notice that



**Figure 5-6**  
Stack in Action (Several Calls).

any subroutine can call any other subroutine. The third column shows what is happening to the stack. The most recent return address is always at the top of the stack. This is called the **top of the stack**, even though the 8085 builds the stack toward a lower address.

If the subroutines are written to allow it, you can even have a subroutine call itself. This is called **recursion**; or when the self-call takes place, the action is said to be **recursive**. To allow recursion, the subroutines must be written so that the activity eventually returns to the higher level. Naturally, if your subroutines are recursive you must have a stack area that is large enough to hold as many returns as the number of calls that might occur.

**Self-Test Review**

6. A part of the program that can be called when needed is called a \_\_\_\_  
\_\_\_\_\_.
7. What two things happen when a call is executed?  
\_\_\_\_\_  
\_\_\_\_\_.
8. What is the mnemonic for a call if carry = 1? \_\_\_\_\_.
9. What is the mnemonic for an unconditional return? \_\_\_\_\_.
10. What flag is checked by the CPE instruction? \_\_\_\_\_.
11. After you return from a subroutine, what should be on the stack?\_\_\_\_  
\_\_\_\_\_.
12. What address is called by RST 7? \_\_\_\_\_.
13. How can you use an RST instruction to call a routine that is more than eight bytes long? \_\_\_\_\_  
\_\_\_\_\_.

## INSTRUCTIONS THAT CHANGE THE STACK

In addition to using the stack to store return addresses for subroutines, you can use the stack to store other data. To permit this, there are several special 8085 instructions. You have already learned about XTHL, which exchanges the value in the HL register with the value on top of the stack. But there are four register pairs in the 8085, and each of them should have some access to the stack. So they do. To place a register pair on the stack, you use a PUSH instruction. To retrieve the value and put it back into the register pair, you use a POP instruction.

One significant thing to notice in all stack activities is that the stack pointer indicates the last address used by a push. For a pop, the first address to be popped is the current stack pointer. For consistency, all stack storage and retrieval operations are performed in two bytes. Therefore, you cannot PUSH B without C. Similarly, you cannot retrieve a 1-byte register without its companion. POP B will get both B and C.

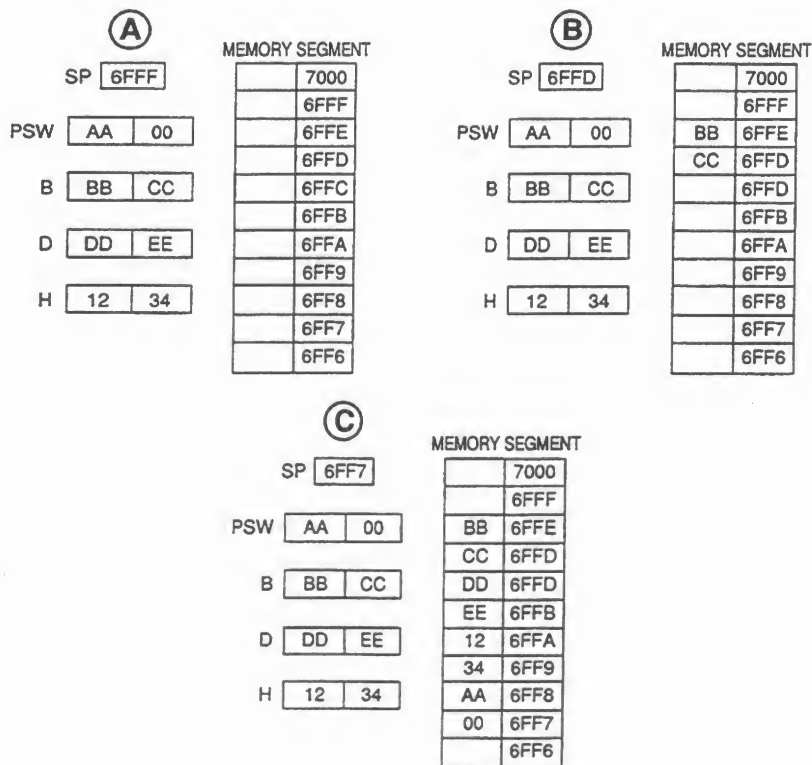
The PUSH and POP instructions work with four register pairs; BC, DE, HL, and PSW. The first three are quite obvious. PSW, as you may recall, is the accumulator and the flags treated as a single sixteen-bit register. The mnemonics and opcodes for the four PUSH and four POP instructions are listed in Figure 5-7.

<u>Mnemonic</u>	<u>Opcode</u>	<u>Mnemonic</u>	<u>Opcode</u>
POP B	C1	PUSH B	C5
POP D	D1	PUSH D	D5
POP H	E1	PUSH H	E5
POP PSW	F1	PUSH PSW	F5

**Figure 5-7**

The PUSH and POP instructions.

The PUSH operation performs the same function for its designated register pair as the CALL operation performs for the next instruction address. The registers are stored on the stack and the stack pointer is decremented by 2. Assume the registers are as shown in Figure 5-8A. If you execute a PUSH B, the values will change to those shown in Figure 5-8B. Notice that the only register that is changed is the stack pointer. This is indicated by the value in the SP register, and the arrow at the side of the memory segment.



**Figure 6-8**  
Stack Operations.

In some instances it is desirable to store all register values on the stack at the beginning of the subroutine and restore them at the end of the subroutine. It is not so important which registers are pushed first. But, it is vital that they be popped in proper order so that each register receives its own value. Figure 5-8C shows the stack with all registers pushed.

After all registers are popped, the stack pointer will again point to address  $6FFF_{16}$  (8 more than the current value). The register values will still be in memory, but they cannot be easily accessed through the stack related instructions. However, if you know how many pushes or pops have taken place, you can adjust the stack pointer to a desired value by using the INX SP and DCX SP instructions. For example, if you INX SP twice and then push an address onto the stack, the next return will go to the pushed value rather than the instruction after the last call. As you can imagine, such stack manipulations can be very confusing and you must do them very carefully.

The LXI SP instruction can be used to set the stack pointer to any predetermined value. However, because this is an immediate operation, the value will always be the same, regardless of when the instruction occurs. For this reason, you must use it with even more caution (if possible) than the INX and DCX instructions.

This leads us to a side discussion about labels in assembly language programs. As you learned earlier, you can use a label to establish the value in an LXI instruction. This is illustrated in Figure 5-9. The ZERO EQU 0 statement assigns a value of 0 to the label ZERO. Similarly, the label BEGIN acquires a value equal to the instruction counter when the LXI H,0 instruction is assembled. What does that mean? The instruction counter for an assembler counts the bytes required for each instruction. You can establish a beginning instruction counter value at the beginning of your program with an ORG statement.

Both the EQU and ORG statements are called **pseudo** operations or **assembler directives**. They do not translate directly to 8085 opcodes, but instead provide information to the assembler. Unlike the opcode mnemonics, which are reasonably standard for 8085 assemblers, the assembler directive mnemonics may vary or simply not exist in different assemblers. Although most of these are explained by the list of common assembler directives in Figure 5-10, the DB, DS, DW, and ORG require special explanation.

	TITLE	"Pseudo Op Demonstration"	
	STL	"Listing control"	
	SPACE	3	
ZERO	EQU	0	;Set ZERO to 0
TRUE	EQU	0	;True is when the byte is 00.
FALSE	EQU	-1	;Just so its not 00.
QUESTION	SET	TRUE	;For this assembly
	ORG	6FFFH	;Start the program at 6FFF Hex
STACK	DS	1	;Top of stack here
BEGIN	LXI	H,0	;Zero out H and L
	DAD	SP	;Copy SP into HL
	SHLD	STKSTR	;Save the stack pointer
	LXI	SP,STACK	;Establish stack
	.	.	
	.	.	;This is where the program goes
	.	.	
	IF	QUESTION	;Start conditional
	.	.	
	.	.	
	.	.	
	ELSE	.	;Do this when QUESTION is false
	.	.	
	.	.	
	.	.	
	ENDIF	.	;End of conditional
	.	.	
	.	.	
	.	.	
	LHLD	STKSTR	;Get SP out of storage
	SPHL	.	;Restore the original stack
STKSTR	DS	2	;Need two bytes here
MSG	DB	'Hello',FFH	;put message in memory
ADD1	DW	7000H	;ADD1 points to a value of 7000
	END	.	;This will have the appropriate ; instruction to return to the ; operating system.

**Figure 5-9**  
Using labels and other assembler directives.



The DB (define byte) statement puts the ASCII values for the word "Hello" followed by FF<sub>16</sub> in memory. During the program, a register pair may be pointed to MSG and the message can be accessed. DS (define space) cause the instruction counter to increment. Thus, the beginning of the program is at 7000H and the stack pointer will have an unused address for the first stack operation. DW defines pairs of bytes. This is useful, because you can write the byte pairs in high-low sequence (7000H) instead of the low-high (00,70) sequence required by the DB instruction.

The ORG, for origin, statement sets the instruction counter in the assembler and determines the actual addresses for the labels used in jumps and calls. Some assemblers use the CSEG and DSEG statements instead of ORG. CSEG and DSEG, which stand for **code segment** and **data segment**, are required for 8088 and the compatible 16 bit microprocessors. The instruction set for the 8085 is a part of the instruction set used for those more advanced microprocessors. As a result, the source code for the 8085 can also be used for the 8088, but only if the right assembler directives are used. As with the other assembler directives, you will have to check the documentation for your assembler to determine the directives you need to know.

CSEG n	Code Segment	Address n is the first code address.
DB b1,b2,etc.	Define Byte	Assigns byte-by-byte values to those locations in memory. Each item (b1, b2, etc.) is a byte value.
DS n	Define Space	Reserves n bytes of memory.
DSEG n	Data Segment	Address n is the first data address.
DW w1,w2,etc.	Define Word	Assigns word-by-word values to those locations in memory. Each item (w1, w2, etc.) is a 2-byte value.
EJECT	Listing control	Causes a form feed to the printer or listing file.
ELSE	Conditional Assembly	Skips following group of mnemonics when previous IF was accepted.
END		Identifies the end of the program.
ENDIF	Conditional Assembly	
EQU	Equals	Assigns a value to a label.
IF		Marks the beginning of conditional mnemonics and establishes the condition.
ORG	Origin	Assigns a value to the assembler's instruction counter.
SET		Assigns a value to a label. Unlike EQU, the label of a SET instruction may be redefined later in the source code.
STL	Subtitle	Establishes a subtitle in the listing.
TITLE	Title	Establishes a title for the listing.

**Figure 5-10**  
Assembler Directives or Pseudo Operations.

And now, a final word on the use of SPHL and XTHL. In some cases it is desirable to use the DAD instructions to compute a value for the stack pointer. The result is in the HL pair, and SPHL is the only way to get a computed stack value into the stack pointer. XTHL is used to adjust a value currently on the stack. You use XTHL to trade the value into the HL pair. You can then use DAD instructions to modify it. Finally, by using XTHL again, you can return the modified value to the stack and restore the old HL value.

### Self-Test Review

14. What instruction puts a register pair onto the stack? \_\_\_\_\_
15. What instruction gets a value from the stack? \_\_\_\_\_
16. What instruction adds one to the value in the stack pointer?  
\_\_\_\_\_
17. What instruction subtracts one from the value in the stack pointer?  
\_\_\_\_\_
18. What assembler directive would you use to put a value of 1234H into memory? \_\_\_\_\_
19. What assembler directive establishes the starting address of the program? \_\_\_\_\_
20. What assembler directive assigns a value to a label? \_\_\_\_\_
21. What is another name for assembler directive? \_\_\_\_\_  
\_\_\_\_\_

## UNIT SUMMARY

1. A stack is a series of temporary storage locations that permit easy input and removal of data.
2. A first in, first out (FIFO) stack is also called a queue.
3. The 8085 stack is last in, first out.
4. In an 8085, the lowest address is the top of the stack.
5. When working with the stack, you must always know what it is doing.
6. You may have to initialize the stack before you can use it.
7. To load the stack pointer into the HL register pair, you must LXI H,0 and then DAD SP.
8. Subroutines are program segments that can be accessed by other parts of the program.
9. When a subroutine can be called from itself, it is said to be recursive.
10. A call (CALL) is a branch to a subroutine that leaves a return address on the stack.
11. When a call is executed 1) the return address is put on the stack, and 2) the PC is given the subroutine address (the argument of the call).
12. Conditional calls allow you to execute a subroutine based on the state of a flag. The conditional calls are as follows:

CNZ	Call Not Zero	Z=0
CZ	Call Zero	Z=1
CNC	Call Not Carry	C=0
CC	Call Carry	C=1
CPO	Call Parity Odd	P=0
CPE	Call Parity Even	P=1
CP	Call Positive	S=0
CM	Call Minus	S=1

## 5-24 STACK OPERATIONS AND SUBROUTINES

---

13. The return statement (RET) at the end of a subroutine pops the address from the stack into the program counter.
14. If a subroutine is called by a conditional call, it need not have a conditional return. The conditional returns are as follows:

RNZ	Return Not Zero	Z=0
RZ	Return Zero	Z=1
RNC	Return Not Carry	C=0
RC	Return Carry	C=1
RPO	Return Parity Odd	P=0
RPE	Return Parity Even	P=1
RP	Return Positive	S=0
RM	Return Minus	S=1

15. An RST is a special call to one of eight predefined addresses.
16. To determine the address called by an RST n instruction, multiply the value of n by 8.
17. A common way to permit an RST subroutine to have more than 8 bytes of code is to put a jump at the RST address.
18. After a subroutine, the stack should be returned to its previous condition.
19. To copy a value from a register pair onto the stack, you use a PUSH instruction.

20. To transfer a value from the stack into a register pair, you use a POP instruction.
21. The four register pairs are BC, DE, HL, and PSW.
22. INX SP will add one to the value in the stack pointer.
23. DCX SP will subtract one from the value in the stack pointer.
24. Assembler directives do not translate to code, but they do provide information for the assembler.
25. Another name for assembler directives is pseudo operation.
26. The five most common assembler directives are as follows:

DB b1,b2,etc.	Define Byte	Assigns byte-by-byte values to those locations in memory. Each item (b1, b2, etc.) is a byte value.
DS n	Define Space	Reserves n bytes of memory.
DW w1,w2,etc.	Define Word	Assigns word-by-word values to those locations in memory. Each item (w1, w2, etc.) is a 2-byte value.
EQU	Equals	Assigns a value to a label.
ORG	Origin	Assigns a value to the assembler's instruction counter.

## **EXPERIMENTS**

Perform Experiments 14, 15, and 16.

*Unit 6*

**INPUT/OUTPUT  
OPERATIONS AND  
INTERRUPTS**



### CONTENTS

INTRODUCTION . . . . .	6-3
UNIT OBJECTIVES . . . . .	6-4
OUTPUT OPERATIONS . . . . .	6-5
INPUT OPERATIONS . . . . .	6-8
SERIAL I/O . . . . .	6-10
Input . . . . .	6-11
Output . . . . .	6-12
INTERRUPTS . . . . .	6-14
More Interrupts . . . . .	6-16
Reset or Restart . . . . .	6-18
UNIT SUMMARY . . . . .	6-20
EXPERIMENTS . . . . .	6-23

## INTRODUCTION

This is the final unit of this 8085 microprocessor programming course. There are two major items yet to be discussed. The first of these is the input and output (I/O) operations. The second is the way the 8085 handles interrupts. Before you read about these subjects, let's briefly review what I/O and interrupts are.

Input and output are the operations by which the MPU communicates with the outside world. That is, I/O operations are any operations except memory addressing and internal register manipulation. While that sounds like a great deal, it really is not. There are only two I/O commands: IN and OUT.

The other instructions you will learn in this unit relate to the 8085 interrupts. The term interrupts is used for both the operations caused by external signals to the MPU and the external signals themselves. Thus, when the MPU receives an interrupt, it performs an interrupt routine.

## UNIT OBJECTIVES

When you complete this unit you will be able to do the following:

1. Describe the effects of the following operations: NOP, RIM, SIM, OUT, IN, DI, and EI.
2. State the difference between the memory addresses and the I/O addresses.
3. State the definition of the word interrupt.
4. Identify the operations that occur when the 8085 receives an interrupt.
5. Identify the activities that might occur when the MPU responds to an interrupt or perform an I/O routine.
6. Identify the terms port, "maskable" interrupt, and "non-maskable" interrupt.
7. List the bits in the 8085 interrupt mask and identify their significance.
8. State the differences between RST, INTR, and TRAP.

## OUTPUT OPERATIONS

The 8085 output operations are performed by the **OUT** opcode, D3<sub>16</sub>. The byte following the opcode is the address of the I/O port. The Assembly language source code for the OUT instruction is simply as follows:

OUT            port

The value of *port* in the 8085 must be between 0 and 255<sub>10</sub>. Just like a memory address, the *port* identifies a location. Circuitry attached to the address bus and the IO/ $\overline{M}$  line detect which I/O address is used.

A port, as you may recall from Unit 1, is the MPU connection to the outside world. The word **port** is used to identify all of the things associated with that connection. The address is the port. The physical connector is the port. Any other hardware required for that connection is called the port.

Usually, there is an integrated circuit (IC) built into the computer at the output address. There is a wide range of IC's that can be used. The most simple devices are buffers that serve to isolate the output from the data bus. This prevents changes on the output from affecting other data bus operations.

Slightly more complicated than a buffer is a latch. The only difference between a buffer and a latch is that the latch retains the value after the OUT command, until another value is output to that port address.

To understand how I/O and memory addressing are different, you need to be aware of three control lines supplied by the 8085 to the computer circuitry. They are listed in Figure 6-1.

<u>Line</u>	<u>Function</u>
$\overline{RD}$	Read operation
$\overline{WR}$	Write operation
$\overline{IO/M}$	I/O vs. Memory operation

**Figure 6-1**  
The I/O and memory control lines.

The  $\overline{RD}$  and  $\overline{WR}$  lines are used for either I/O or memory.  $\overline{RD}$  is held low for any read operation, and  $\overline{WR}$  is held low for any write operation. The  $IO/\overline{M}$  line provides a difference between I/O addresses and memory addresses. When the  $IO/\overline{M}$  line is high, the I/O address is selected. When the  $IO/\overline{M}$  line is low, memory is selected. Of course this requires that the hardware be connected to these lines as well as the address lines. However, I/O requires only the bottom eight address lines, because the highest I/O address is  $255_{10}$ .

A further distinction is that output operations are always from the accumulator, while memory access can also be made from the B, C, D, E, H, or L registers. As a result, you must always **place the value** to be output **in the accumulator** before executing the OUT opcode.

In some cases, the I/O activity requires several port addresses. When this occurs, one address (normally the lowest address) is called the **base port address**. For example, if a particular I/O port control IC requires eight (8) addresses, the base port might be  $B0_{16}$ . The base address would probably be the one you send data to. Above that you would have other inputs and outputs such as the control and status registers for that IC. The output to that port would then require not only outputting the data, but also the control bytes to those addresses. It might also be necessary to input the status to determine if the device is ready to accept the data.

In assembly language, it is customary to assign a label to each specific I/O port or base port. This is done at the beginning of the program, so that the label is easy to locate and change if necessary. Most assemblers allow you to refer to the other addresses by providing the label and an offset. The example in Figure 6-2 shows how this works.

	ORG	7000H	;Start the program at address 7000 hex
PORT	EQU	0B0H	;LCD port
BEGIN			;Program begins here
			;And continues through here
	MVI	A,03H	;Put control code in accumulator
	OUT	PORT	;Send control code to port
	MVI	A,33H	;Put data code in accumulator
	OUT	PORT+1	;Send data to display (port + 1)
			;More program here
	END		; And then we are done.

**Figure 6-2**  
Using labels with offsets.

Notice in Figure 6-2 that the value PORT is equal to B0<sub>16</sub> as set by the EQU pseudo operation. Then, during the program, the control port, B0<sub>16</sub>, is referred to by the label PORT. The data port, B1<sub>16</sub>, is identified by the expression PORT+1.

It is quite common to have to send a control code, check a status value, or both before you can output a value. To check a status value, you would use the IN opcode as described in the following.

## INPUT OPERATIONS

Input operations are similar to output operations, except that the value is received by the accumulator instead of being output from the accumulator. It is therefore important to be aware that the accumulator value will be changed by an IN operation. The value formerly in the accumulator will be replaced by the value at the input port.

Aside from that, the IN instruction (opcode DB<sub>16</sub>) is the same as the OUT. Physically though, there is a difference. When the OUT instruction occurs, the  $\overline{WR}$  line is brought low. The hardware detects this to decode the address. For an IN operation, the  $\overline{RD}$  line is brought low. This is also detected by the address decoding circuitry. Because these are two separate signals, it is not necessary for the input and output ports to be related to each other. In other words, if input port C0<sub>16</sub> is the keyboard, output port C0<sub>16</sub> could be the display. These two devices are often thought of together, but they are not physically related.

This is probably the single most important thing to remember. Just because you write a value to an output port, does not mean you can read that value from the same port. The device associated with those two ports may be entirely different.

By the same token, it may be possible to have the output to a device at one port and the input from the same device at a different port. The result could be that you can input through one port the same data that was output to another. However, it is usually most convenient to use the same port for both input and output to each specific device. It not only makes it easier to decode in the hardware, it also makes it easier to prepare the software. Because the ports are the same, you can use the same label to define input and output.

Figure 6-3 is an example, where input PORT+1 is the status port and bit 2 indicates that data is ready to be received. Output PORT+1 bit 2 is a response to the other device that we are ready to receive data. The data is received at input address PORT. Because this is a hypothetical computer, the actual PORT value has been omitted.

OK	EQU	4H	;BIT 2 (0000 0100) SET MEANS OK
NOTOK	EQU	0	;CLEAR ALL BITS WHEN IT IS NOT OK
	ORG	100H	
BEGIN			
	MVI	A,OK	;SET UP AN OK
	OUT	PORT+1	;SEND IT OUT
LOOP	IN	PORT+1	;INPUT STATUS
	ANI	4H	;MASK THE OK BIT
	JZ	LOOP	;LOOP UNTIL READY
	MVI	A,NOTOK	;SET UP A NOT OK
	OUT	PORT+1	;TELL THEM TO WAIT
	IN	PORT	;GET THE DATA
	MOV	M,A	;STORE THE DATA IN MEMORY
	END		

**Figure 6-3**

A Hypothetical Input/Output Exchange.

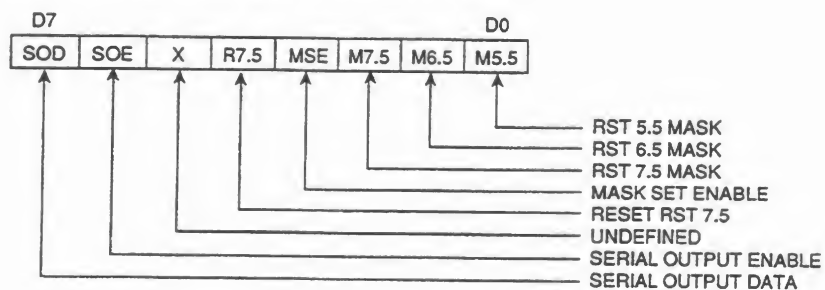
In this routine, the OK is first sent out to PORT+1 saying we are ready to receive data. Then, the status bit is sampled to see if the data is ready. If it is not, another sample is taken. This LOOP is repeated until there is data available. Immediately after the data is available, the NOTOK signal is sent so that no new data is sent. Then, the data is retrieved and placed in memory. This process would be repeated until some signal (either in the data or in the status) indicated that the message was complete. Because all eight bits of data are received at the same time, this is called a parallel input.



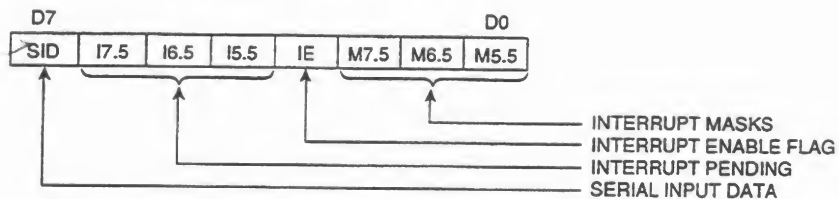
## SERIAL I/O

Serial data transfer takes place one bit at a time. The 8085 has a special way of performing serial input and output. There are two connections to the microprocessor identified as SID (serial input data) and SOD (serial output data). These two lines are accessed through the RIM and SIM instructions. A thorough description of these instructions is provided in the section on interrupts. For now it is only important to understand how the instructions control serial I/O. Figure 6-4 shows the accumulator values before a SIM instruction and after a RIM instruction.

A REGISTER BEFORE EXECUTING SIM



A REGISTER AFTER EXECUTING RIM



**Figure 6-4**  
Accumulator before SIM and after RIM.

## Input

The values after a RIM (Read Interrupt Mask) are easiest to understand. The high bit is the value that was on the SID line when the RIM was executed. That value can be used by the other 8085 instructions. For example, if you want to input an entire 8-bit value, you could perform a sequence similar to the one in Figure 6-5.

	MVI	B,0	;Clear the B register
	MVI	C,7	;Establish a counter
LOOP			;Repeat this 7 times
	CALL	WAIT	;Wait for next input
	RIM		;Read the interrupt mask
	ANI	80H	;AND with 80 <sub>16</sub> to mask the high bit
	ORA	B	;OR with B
	RRC		;Rotate one place to the right
	MOV	B,A	;Save the result in B
	DCR	C	;Decrement counter
	JNZ	LOOP	;Repeat until through
ONE2GO			;Read last bit
	CALL	WAIT	;Wait for next input
	RIM		;Read the interrupt mask
	ANI	80H	;AND with 80 <sub>16</sub> to mask the high bit
	ORA	B	;OR with B

**Figure 6-5**

Reading a byte from the serial port.

This is a straight forward routine, except that it ignores what happens during the WAIT routine, which is not listed. To be sure your are not reading the same bit twice, there must be some timing consideration. This is normally controlled by the interrupts, either on a specified time interval or through an external stimulus.

The other values read by the RIM opcode are the interrupt pending bits, the interrupt enable bit, and the interrupt mask. These bits may have nothing to do with the serial input, or they may indicate the timing. The situation is determined by the hardware and software of the computer system you happen to be using.

To understand these other bits better, you must learn about the four maskable interrupts. They are called RST 7.5, RST 6.5, RST 5.5, and INTR. If any of the first three have occurred, the corresponding bit will be set (1). This information can be used to determine what course of action the processor should take. The low four bits of the interrupt mask indicate which of the four interrupts are permitted. If bit 3, IE, is a one, it means the INTR interrupt can be accepted. The low three bits M7.5, M6.5, and M5.5 correspond to the other three maskable interrupts.

## Output

To set or clear the interrupt mask, you use the SIM (Set Interrupt Mask) opcode. This will be covered in detail when the interrupts are described. For now, it is only necessary to worry about the two highest bits, bit 6 and bit 7. Bit 6 is called **serial output enable** (SOE), and bit 7 is **serial output data** (SOD). To make the SOD line from the 8085 to go high, you must set both bits 6 and 7. To make the SOD line go low, you must keep bit 6 high and reset bit 7. If bit 6 is low, the SOD line will not change state, regardless of the status of bit 7.

The serial output, just like the input, must be timed to correspond to the requirements of the receiving device. This can be done through interrupts from the receiving device or from a timer. In the early days of microprocessors, it was not uncommon to use timing loops to control the length of time between signal transitions. To do this, the time required for each instruction was carefully determined and the sequence repeated the required number of times between outputs (or inputs).

One instruction that is useful for adding small increments of time is the NOP instruction. NOP stands for no operation, and that is just what it does—nothing. Another use for NOP is to fill a place in memory that contains an unwanted instruction. That can save you a lot of time when you are modifying a portion of a program and don't wish to reassemble and reload the whole program.

The use of timing loops has fallen into disuse, because it is often desirable to use the microprocessor for other activities while waiting the prescribed time. Another reason is that by using an external timing source, the precise number of instructions and their timing is not critical. Therefore, if a faster microprocessor is used, the code for serial I/O can remain the same.

## Self-Test Review

1. What is identified by the byte following the OUT opcode?\_\_\_\_\_
2. What data (where is it located) is output by the OUT opcode?\_\_\_\_\_
3. What control line determines whether an operation is working with an I/O port or a memory address?\_\_\_\_\_
4. Must the  $\overline{IO/\overline{M}}$  line be high or low for an I/O operation?\_\_\_\_\_
5. What control line must be low to do a read operation.\_\_\_\_\_
6. What does the mnemonic RIM stand for?\_\_\_\_\_
7. Which bit contains the serial input data after a RIM instruction is executed?\_\_\_\_\_
8. What does the mnemonic SIM stand for?\_\_\_\_\_
9. Which bits must contain the serial output data before a SIM instruction is executed?\_\_\_\_\_
10. What type of data transfer is made eight bits at a time?\_\_\_\_\_

## INTERRUPTS

Closely associated to input and output are microprocessor interrupts. The 8085 has five hardware interrupt inputs: INTR, RST 5.5, RST 6.5, RST 7.5, and TRAP. The three RST signals and the INTR signals can be masked, or disabled. TRAP is non-maskable and has the highest priority. However, no interrupt is processed until the current opcode has completed its activity.

Interrupt priorities determine which interrupt routine to perform when more than one interrupt occurs. The priorities are listed in Figure 6-6. If several interrupts occur at once, the highest priority will be accepted. To prevent another interrupt from occurring during an interrupt routine, most interrupt routines mask the other interrupts until the routine is completed. As a result, only a TRAP interrupt, which cannot be masked, can interfere with the execution of the interrupt routine. In all cases, the PC is pushed onto the stack before the interrupt subroutine is executed.

<u>Name</u>	<u>Priority</u>	<u>Destination Address</u>	<u>Trigger Type</u>
TRAP	1	24H	Rising edge and High level until sampled
RST 7.5	2	3CH	Rising edge (latched)
RST 6.5	3	34H	High Level until Sampled
RST 5.5	4	2CH	High Level until Sampled
INTR	5	See note	High Level until Sampled
Note:	The address is established by the interrupting device.		

**Figure 6-6**  
Interrupt Priority Table.

Also listed in Figure 6-6 are the destination addresses for each interrupt. The addresses for the RST interrupts are easy to remember, because they can be computed from the name, in the same manner as the addresses for the software RST instructions. For example, the destination for RST 5.5 is 5.5 times 8, or  $44_{10}$  ( $2C_{16}$ ). Similarly, the destination for RST 7.5 is 8 times 7.5, or  $60_{10}$  ( $3C_{16}$ ). You can also think of TRAP as TRAP 4.5, because that is the interrupt destination ( $24_{16}$ ).

Although the RST and TRAP interrupts perform as though they are executing a restart (RST) instruction, the instruction is not read from memory. It is part of the microprocessor. The INTR instruction performs as though it is fetching an instruction. But instead of reading memory, it just reads the data bus. Whatever value is there is used as the next instruction. The 8085 depends on the interrupting device to supply the proper RST (or other) instruction. The INTA signal from the 8085 must be read by the interrupting device and that device must supply an instruction to the data bus.

The 8085 was designed to be upward compatible with the 8080 MPU. As a result, there are different instructions to enable or disable the INTR and RST interrupts. The 8080 had only one maskable interrupt, INT. This corresponds to the INTR interrupt in the 8085. So that the same code can be used for both chips, the 8085 supports the EI and DI opcodes to enable and disable the INTR line. DI disables the interrupt, and EI enables it. As you learned previously, you can determine whether the INTR interrupt is enabled by checking bit 3 after a RIM instruction.

To enable, or disable, the RST interrupts you supply a value in the accumulator and execute a SIM instruction. When clear, bits 0, 1, and 2 enable RST 5.5, RST 6.5, and RST 7.5 respectively. To change the interrupt mask, you must also set bit 3. This allows you to work with the serial output through bits 6 and 7, without changing the interrupt mask.

Recall that no interrupt is processed until the current opcode has completed its activity. However, RST 7.5 is supposed to be triggered by a rising signal on the RST 7.5 line. Because the service routine cannot be started until the current instruction is done, it is possible for the RST 7.5 signal to go low again before it is answered. To prevent this, the 8085 has an internal latch that holds the RST 7.5 after a rising edge. This latch will retain the RST 7.5 until it is cleared. You clear the latch by setting bit 4 of the accumulator and executing a SIM instruction. If the RST 7.5 mask bit is still clear, this action reenables the RST 7.5 line.

Bits 0, 1, and 2 are also used when you read the interrupt mask. By checking these bits, you can see which of the RST interrupts are enabled, and which ones are disabled. You should think of the interrupt mask as a special input and output port. Notice that the three low bits of the input correspond in functions to the output. In fact, the input values are determined by the output of latches in the 8085. You set those bits by outputting values with the SIM instruction, and can then read the latches with the RIM instruction. This is easier to understand, when you think of the EI and DI instructions. EI and DI set or clear the INTR interrupt mask. You can determine whether it is set or cleared by checking bit 3 after a RIM instruction.

RST 7.5 is not only the highest priority (except TRAP and RESET), it has the highest starting address. So what? You might ask. Because it has the highest starting address, there is a large amount of memory available for its routine. You do not need to use a jump to allow for additional space. This was probably one of the reasons it was selected to have the highest priority.

## More Interrupts

Often, it is desirable to have more than five devices causing interrupts. This is easily done, if you associate the interrupts with an I/O port. You know that the microprocessor can determine which type of interrupt occurred by reading the interrupt mask and seeing which interrupts are waiting (bits 4, 5, and 6). You can do the same thing with each interrupt signal. Have each interrupting device control a certain bit of an input port. Then when the interrupt occurs, your interrupt routine can read that port and determine which device caused the interrupt.

From this you may have noticed a pattern in responding to interrupts. There are six principal steps as follows:

1. Mask out any other interrupts by clearing the mask and using the DI instruction.
2. Store the registers.
3. Determine the cause of the interrupt.
4. Respond to the cause of the interrupt.
5. Restore the registers.
6. Reenable the interrupts by setting the mask and using the EI instruction.

Let's examine each of these steps in more detail.

The first step is to mask out other interrupts.

The second step is to store the register values. You learned how to do this in the last unit. All that is usually required is a quick series of PUSH instructions. However, you must perform part of this step before you complete the first step because you cannot put the mask into the accumulator and set it until the accumulator has been stored. Therefore, you begin your interrupt service routine with DI to prevent the INTR signal from getting in. Next, PUSH PSW so you can use the accumulator. Then you can set up your interrupt mask and execute a SIM instruction. After that you will have time to push the remaining registers.

The third step is to identify the cause of the interrupt. Unless more than one device is attached to the interrupt line, this is determined by the destination address. If more than one interrupt is attached to a given line, the routine for that interrupt will have to read some value to determine the cause.

Fourth, the routine must respond to the cause of the interrupt. This is the part of your routine that "takes care" of what has happened. Interrupts are used for many things. If the interrupt was from the keyboard, you may have to store the value in a buffer so that it can be used later, or you may need to perform some operation immediately. If the interrupt is from a timer, you may update a clock value, receive the next serial bit, or perform some other needed service. If the interrupt is from a power detector, it may be necessary to close all I/O files immediately. In such cases, the second step (saving the registers) might not be needed.

Fifth, after the cause of the interrupt has been attended, your routine can get ready to return to its normal operation. To return to normal, all the registers must be restored to their previous values. This is the fifth part of an interrupt service routine.

With the registers properly restored, there is a sixth and final step, your routine should make before it returns control to the normal program. The interrupts must be reenabled by setting the interrupt mask and executing an EI instruction. Of course it is quite possible that once the interrupts are reenabled another interrupt may take control of the microprocessor's operation. If so, the whole routine may be repeated.



## Reset or Restart

The reset signal is a special interrupt. It resets the instruction pointer to  $0000_{16}$ , but it does not store any values on the stack. Therefore, the reset is a **JMP** to zero, and not a **CALL**. This poses a slight problem in regard to the **RST 0** instruction, which also has a destination of address 0. To return from the **RST 0**, the subroutine should use an **RET** opcode. But because the **RESET** does not put a return address on the stack, the destination of a **RET** instruction might be anything. How is an **RST 0** instruction different from the hardware reset?

Operationally, the only difference is that the **RST** pushes a return address onto the stack. To allow for this, you must make some test before executing a return from this routine. The simplest approach is to not provide a return, and say that you cannot use **RST 0**. Because the **RESET** routine reinitializes the stack, this is the approach used in most 8085 computers. The effect is that executing **RST 0** does the same thing as pressing the reset button. This is considered the best approach.

The only true alternative is to make a test to determine the conditions. This is difficult, because you cannot check the stack pointer to see if it has changed from its original value. However, the stack pointer value is not established by the microprocessor on reset, but by the routine, therefore you cannot say what value it will have on power-up. It might happen to match a valid value. This is true for all internal registers.

The approach used by some manufacturers is to provide a register, or latch, at a certain I/O port, that is cleared when the reset signal occurs. If this port contains any value but zero, you can assume the computer was operational and the restart at zero was not a reset.

## Self-Test Review

11. What are the five hardware interrupt signals to the 8085?  
\_\_\_\_\_
12. What is the destination of the RST 7.5 interrupt?\_\_\_\_\_
13. How much room is there between the beginning of the RST 7.5 service routine and the beginning of the next fixed destination interrupt routine?\_\_\_\_\_
14. Where does the 8085 look to find the instruction after an INTR?  
\_\_\_\_\_
15. What is the first thing the interrupt service routine should do?  
\_\_\_\_\_

## UNIT SUMMARY

1. The 8085 **OUT** opcode, D316, requires two bytes. The second byte is the I/O port.
2. The 8085 **IN** opcode, DB16, requires two bytes. The second byte is the I/O port.
3. 8085 I/O ports have addresses from 0 and 255<sub>10</sub>.
4. For I/O addressing, the  $\text{IO}/\overline{\text{M}}$  line must be high.
5. The three 8085 I/O and memory control lines are as follows:

<u>Line</u>	<u>Function</u>
$\overline{\text{RD}}$	Read operation
$\overline{\text{WR}}$	Write operation
$\text{IO}/\overline{\text{M}}$	I/O vs Memory operation

6. I/O operations are always to or from the accumulator.
7. A single address used to identify a group is called the **base port address**.
8. The 8085 can perform serial I/O through the SID (serial input data) and SOD (serial output data) lines.
9. The SID and SOD lines are accessed by the RIM and SIM instructions.
10. RIM reads the following values to the accumulator:

<u>Bit</u>	<u>Value</u>
0	Mask for RST 5.5
1	Mask for RST 6.5
2	Mask for RST 7.5
3	Interrupt Enable (INTR mask)
4	Pending RST 5.5
5	Pending RST 6.5
6	Pending RST 7.5
7	Serial input data (SID)

11. SIM sets the following values from the accumulator:

<u>Bit</u>	<u>Value</u>
0	Mask for RST 5.5
1	Mask for RST 6.5
2	Mask for RST 7.5
3	Mask set Enable (RST masks)
4	Reset RST 7.5
5	(Undefined)
6	Enable SOD
7	Serial output data (SOD)

12. NOP is a do nothing opcode that takes up time and space.
13. The 8085 has five interrupt inputs with the following priorities, destinations, and characteristics:

<u>Name</u>	<u>Priority</u>	<u>Destination Address</u>	<u>Trigger Type</u>
TRAP	1	24H	Rising edge and High level until sampled
RST 7.5	2	3CH	Rising edge (latched)
RST 6.5	3	34H	High Level until Sampled
RST 5.5	4	2CH	High Level until Sampled
INT R	5	See note	High Level until Sampled

Note: The address is established by the interrupting device.

14. The destination for RST 5.5, RST 6.5, and RST 7.5 is the RST number times 8.
15. You can also think of TRAP as TRAP 4.5, because of its destination ( $24_{16}$ ).
16. INTR fetches an instruction from the data bus.
17. The INTA signal from the 8085 is read by the interrupting device. That device must supply the instruction to the data bus.
18. EI and DI opcodes enable and disable the INTR line.

19. Interrupt service routines contain the following six principal steps.
  1. Store the registers.
  2. Mask out any other interrupts by clearing the mask and using the DI instruction.
  3. Determine the cause of the interrupt.
  4. Respond to the cause of the interrupt.
  5. Restore the registers.
  6. Reenable the interrupts by setting the mask and using the EI instruction.
20. The reset signal is a special interrupt, that resets the instruction pointer to  $0000_{16}$ .
21. Reset does not store any values on the stack.
22. You must be careful when using RST 0, which has the same destination as reset, because the reset routine cannot have a return instruction.

## **EXPERIMENTS**

You are now ready to perform Experiment 17, 18, and 19.



*Appendix A*

**The 8085 Instruction Set**



## Opcodes and Pseudo Opcodes

### Symbols and Abbreviations

The following symbols and abbreviations are used in this appendix.

<u>SYMBOL</u>	<u>MEANING</u>
accumulator	Register A
addr	a 16-bit address quantity
data	an 8-bit data quantity
data 16	16-bit data quantity
byte 2	the second byte of the instruction.
byte 3	the third byte of the instruction.
port	the 8-bit port address
r, r1, r2, etc	One of the registers A, B, C, D, E, H, or L.
rrr	the binary pattern designating an 8-bit register. 111 = A 000 = B 001 = C 010 = D 011 = E 100 = H 101 = L 110 (not allowed) Used in M instruction.
rp	One of the register pairs: B means registers B and C, with B as the high byte. D means registers D and E, with D as the high byte. H means registers H and L, with H as the high byte. PSW means registers A and the flags, with A as the high byte. SP means the stack pointer. PC means the program counter.
rh	the first (high order) register of a designated pair.
rl	the second (low order) register of a designated pair.
RR	the binary pattern designating a 16-bit register pair. 00 = B 01 = D 10 = H 11 = SP
PC	the 16-bit program counter register (PCH and PCL are used to refer to the high-order and low-order bytes, respectively.)
SP	the 16 bit stack pointer register (SPH and SPL are used to refer to the high-order and low-order bytes, respectively.)
r <sub>m</sub>	bit m of the register r (bits are numbered from low to high 0 through 7, normally left to right)

The condition flags:

AC	Auxiliary carry bit 4 of the Flag Byte.
CY	Carry bit 0 of the Flag Byte.
P	Parity bit 2 of the Flag Byte.
S	Sign bit 7 of the Flag Byte.
Z	Zero bit 6 of the Flag Byte.
( )	The contents of the memory location or registers enclosed in parentheses.
←	"Is transferred to"
∧	Logical AND
⊕	Exclusive OR
∨	Inclusive OR
+	Addition
-	Two's complement subtraction
*	Multiplication
↔	"Is exchanged with"
	The one's complement (e.g. $\bar{A}$ )
n	The restart number 0 through 7
NNN	The binary representation 000 through 111 for restart number 0 through 7, respectively.

## Alphabetical Listing of Opcodes and Pseudo Opcodes (Assembler Directives)

The assembler directives in this appendix are typical of those used by 8085 assemblers. The particular assembler you are using will most likely have some that are not listed and also not have some of those that are listed. However, these functions are typical of the types of directives you will want to use.

**ACI data** (Add immediate with carry) 1100 1110 (CE<sub>16</sub>)

$(A) \leftarrow (A) + (\text{Byte } 2) + (CY)$

The content of the second byte of the instruction and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**ADC M** (Add memory with carry) 1000 1110 (8E<sub>16</sub>)

$(A) \leftarrow (A) + ((H)(L)) + (CY)$

The contents of the memory location whose address is contained in the H and L registers and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**ADC r** (Add register with carry) 1000 1rrr (88<sub>16</sub> + rrr)

$(A) \leftarrow (A) + (r) + (CY)$

The content of register r and the content of the CY flag are added to the contents of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**ADD M** (Add memory) 1000 0110 (86<sub>16</sub>)

$(A) \leftarrow (A) + ((H)(L))$

The content of the memory location whose address is contained in the H and L registers is added to the contents of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

<b>ADD r</b>	(Add register)	1000 0rrr ( $80_{16} + rrr$ )
--------------	----------------	-------------------------------

$$(A) \leftarrow (A) + (r)$$

The content of register r is added to the contents of the accumulator.  
The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

<b>ADI Data</b>	(Add immediate)	1100 0110 (C6 <sub>16</sub> )
-----------------	-----------------	-------------------------------

$$(A) \leftarrow (A) + (\text{byte } 2)$$

The content of the second byte of the instruction is added to the contents of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

ANAM (AND memory) 1010 0110 (A6<sub>16</sub>)
$$(A) \leftarrow (A) \wedge ((H)(L))$$

The content of the memory location whose address is contained in the H and L registers is logically ANDed with the contents of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**

Flags: Z,S,P,CY,AC

<b>ANAr</b>	(AND register)	1010 0rrr (A0 <sub>16</sub> + rrr)
-------------	----------------	------------------------------------

$$(A) \leftarrow (A) \wedge (r)$$

The content of register r is logically anded with the content of the accumulator. The result is placed in the accumulator. **The CY flag is cleared.**

Flags: Z,S,P,CY,AC

ANI data (AND immediate) 1110 0110 (E6<sub>16</sub>)
$$(A) \leftarrow (A) \wedge (\text{byte } 2)$$

The content of the second byte of the instruction is logically anded with the contents of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**CALL addr** (Call)

1100 1101 (CD<sub>16</sub>)

$(SP) - 1 \leftarrow (PCH)$

$(SP) - 2 \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by 2. Control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction.

Flags: none

**CC addr** (Carry call)

1101 1100 (DC<sub>16</sub>)

IF (CY)=1

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the carry flag is set, the actions specified in the CALL instruction are performed; otherwise, control continues sequentially.

Flags: none

**CM addr** (Minus call)

1111 1100 (FC<sub>16</sub>)

IF (S)=1

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the sign flag is set, the actions specified in the CALL instruction are performed; otherwise, control continues sequentially.

Flags: none

**CMA** (Complement accumulator) 0010 1111 (2F<sub>16</sub>)

$$(A) \leftarrow \overline{(A)}$$

The contents of the accumulator are complemented (zero bits become 1, one bits become 0). **No flags are affected.**

Flags: none

**CMC** (Complement carry) 0011 1111 (3F<sub>16</sub>)

$$(CY) \leftarrow \overline{(CY)}$$

The CY flag is complemented. **No other flags are affected.**

Flags: CY

**CMP M** (Compare memory) 1011 1110 (BE<sub>16</sub>)

$$(A) - ((H)(L))$$

The content of the memory location whose address is contained in the H and L registers is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A)=((H)(L)). The CY flag is set to 1 if (A)<((H)(L)).**

Flags: Z,S,P,CY,AC

**CMP r** (Compare register) 1011 1rrr (B8<sub>16</sub> + rrr)

$$(A) - (r)$$

The content of register is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if (A)=(r). The CY flag is set to 1 if (A)<(r).**

Flags: Z,S,P,CY,AC

**CNC addr** (No carry call)

1101 0100 (D4<sub>16</sub>)

IF (CY)=0

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the carry flag is not set, the actions specified in the **CALL** instruction are performed; otherwise, control continues sequentially.

Flags: none

**CNZ addr** (Not zero call)

1100 0100 (C4<sub>16</sub>)

IF (Z)=0

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the zero flag is not set, the actions specified in the **CALL** instruction are performed; otherwise, control continues sequentially.

Flags: none

**CP addr** (Plus call)

1111 0100 (F4<sub>16</sub>)

IF (S)=0

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the sign flag is not set, the actions specified in the **CALL** instruction are performed; otherwise, control continues sequentially.

Flags: none

**CPE addr** (Parity even call) 1110 1100 (EC<sub>16</sub>)

IF (P)=1

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the parity flag is set, the actions specified in the CALL instruction are performed; otherwise, control continues sequentially.

Flags: none

**CPI data** (Compare immediate) 1111 1110 (FE<sub>16</sub>)

$(A) - (\text{byte } 2)$

The content of the second byte of the instruction is subtracted from the accumulator. The accumulator remains unchanged. The condition flags are set as a result of the subtraction. **The Z flag is set to 1 if  $(A)=(\text{byte } 2)$ . The CY flag is set to 1 if  $(A)<(\text{byte } 2)$ .**

Flags: Z,S,P,CY,AC

**CPO addr** (Parity odd call) 1110 0100 (E4<sub>16</sub>)

IF (P)=0

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the parity flag is not set, the actions specified in the CALL instruction are performed; otherwise, control continues sequentially.

Flags: none



## A-10 The 8085 Instruction Set

---

**CZ addr** (Zero call) 1100 1100 (CC<sub>16</sub>)

IF (Z)=1

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow (\text{byte 3})(\text{byte 2})$

If the zero flag is set, the actions specified in the CALL instruction are performed; otherwise, control continues sequentially.

Flags: none

**DAA** (Decimal adjust accumulator) 0010 0111 (27<sub>16</sub>)

The eight-bit number in the accumulator is adjusted to form two 4-bit Binary-Coded-Decimal digits by the following process:

1. If the value of the least significant 4 bits of the accumulator is greater than 9 **or** if the AC flag is set, 6 is added to the accumulator.
2. If the value of the most significant 4 bits of the accumulator is greater than 9 **or** if the CY flag is set, 6 is added to the most significant 4 bits of the accumulator.

Flags: Z,S,P,CY,AC

**DAD rp** (Add register pair to H and L) 00RR 1001 (09<sub>16</sub> + RR\*16)

$(H)(L) \leftarrow (H)(L) + (rh)(rl)$

The content of register pair **rp** is added to the content of register pair H and L. The result is placed in register pair H and L. **NOTE: Only the CY flag is affected.** It is set if there is a carry out of the double precision add; otherwise it is cleared.

Flags: CY

**DB** (Define byte) Assembler Directive

The DB pseudo defines byte contents. The DB pseudo is of the form:

Label DB iexp1,...,iexpn

The integer expressions iexp1 through iexpn are expressions which evaluate to 8-bit values. For the DB pseudo, a long string can be substituted for an expression. The long string is a character string of one or more characters delimited by single quotes (') or double quotes ("). You can put a single quote mark (or one used as an apostrophe) in a string enclosed by double quotes, or by putting two single quote marks together. Each of the expressions is converted into an 8-bit binary number and stored in sequential memory locations.

**DCR M** (Decrement memory) 0011 0101 (35<sub>16</sub>)

$((H)(L)) \leftarrow ((H)(L)) - 1$

The content of the memory location whose address is contained in the H and L registers is decremented by one. NOTE: All condition flags **except CY** are affected.

Flags: Z,S,P,AC

**DCR r** (Decrement register) 00rr r101 (05<sub>16</sub> + rrr\*8)

$(r) \leftarrow (r) - 1$

The content of register r is decremented by one. NOTE: All condition flags **except CY** are affected.

Flags: Z,S,P,AC

**DCX rp** (Decrement register pair) 00RR 1011 (0B<sub>16</sub> + RR\*16)

$(rh)(rl) \leftarrow (rh)(rl) - 1$

The content of register pair rp is decremented by one. NOTE: **No condition flags are affected**

Flags: none

**DI** (Disable interrupt) 1111 0011 (F3<sub>16</sub>)

The INTR interrupt is disabled **immediately following the execution of the DI instruction.**

Flags: none

**DS** (Define space) Assembler Directive

The define space pseudo (DS) reserves a block of memory during assembly. This pseudo may be used to set up a buffer area or to define any other storage area. The DS pseudo causes the assembler to reserve a number of bytes.

**DW** (Define word) Assembler Directive

The define word pseudo (DW) defines word constants. Data words are **2-byte** values which are placed into memory space, low-order byte first. NOTE: Strings greater than two characters long are not allowed when you are using the DW pseudo.

**EI** (Enable interrupt) 1111 1011 (FB<sub>16</sub>)

The INTR interrupt is enabled **following the execution of the EI instruction.**

Flags: none

**EJECT** (Listing control) Assembler Directive

The EJECT pseudo causes a new page to be started. When the assembler processes an EJECT pseudo, the output device is instructed to move to the start of a new page during the listing.

**ELSE** (Conditional assembly) Assembler Directive

The ELSE pseudo toggles the state of the assembly conditions. If the conditional assembly flag is set to skip assembling source code, it is changed so source code is now enabled. If lines of source code prior to encountering the ELSE pseudo are being assembled, those following the ELSE pseudo are skipped until an ELSE, ENDIF, or END is encountered. NOTE: The ELSE segment must appear after an IF statement, but before the associated ENDIF statement. (See IF and ENDIF.)

**END** (End program) Assembler Directive

The END pseudo indicates the end of the program. If the END statement is missing, the assembler generates one.

**ENDIF** (Conditional assembly) Assembler Directive

The ENDF pseudo indicates the end of a block of source code designated for conditional assembly. Assembly resumes regardless of the current assembly state (assembling or skipping) when the ENDF conditional assembly pseudo occurs.

**EQU** (Define label) Assembler Directive

The Equate pseudo (EQU) is used to assign an arbitrary value to a symbol. This label may not be redefined by subsequent use as a label in any other statement.

**HLT** (Halt) 0111 0110 (76<sub>16</sub>)

The processor is stopped. The registers and flags are unaffected. The MPU will resume execution when it receives an interrupt or reset. HLT causes a wait condition.

Flags: none

**IF** (Conditional assembly) Assembler Directive

The IF pseudo conditionally disables assembly of any statement following the IF pseudo operator. (See also ELSE and ENDF.)

**IN port** (Input) 1101 1011 (DB<sub>16</sub>)

(A) ← (data)

The data placed on the eight bit bidirectional data bus by the specified port is moved to register A.

Flags: none

**INR M** (Increment memory) 0011 0100 (34<sub>16</sub>)

$((H)(L)) \leftarrow ((H)(L)) + 1$

The content of the memory location whose address is contained in the H and L registers is incremented by one. NOTE: All condition flags **except CY** are affected.

Flags: Z,S,P,AC

**INR r** (Increment register) 00rr r100 (04<sub>16</sub> + rrr\*8)

$(r) \leftarrow (r) + 1$

The content of register r is incremented by one. NOTE: All condition flags **except CY** are affected.

Flags: Z,S,P,AC

**INX rp** (Increment register pair) 00RR 0011 (03<sub>16</sub> + RR\*16)

$(rh)(rl) \leftarrow (rh)(rl) + 1$

The content of register pair rp is incremented by one. NOTE: **No condition flags are affected**

Flags: none

**JC addr** (Carry jump) 1101 1010 (DA<sub>16</sub>)

IF (CY)=1

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the carry flag is set, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction: otherwise, control continues sequentially.

Flags: none

**JM addr** (Minus jump) 1111 1010 (FA<sub>16</sub>)

IF (S)=1

$(PC) \leftarrow (\text{byte } 3)(\text{byte } 2)$

If the sign flag is set, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction: otherwise, control continues sequentially.

Flags: none

$$(\text{PC}) \leftarrow (\text{byte 3})(\text{byte 2})$$

Flags: none

IF (CY)=0

If the carry flag is not set, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction; otherwise, control continues sequentially.

Flags: none

IF (Z)=0

If the zero flag is not set, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction: otherwise, control continues sequentially.

Flags: none

IF (S)=0

If the sign flag is not set, control is transferred to the instruction whose address is specified in byte 3 and byte 2 of the current instruction: otherwise, control continues sequentially.

Flags: none

---

Flags: none

Flags: none

Flags: none

Flags: none

**LDAX rp** (Load accumulator indirect) 000R 1010 ( $0A_{16} + R*16$ )

$(A) \leftarrow ((rp))$

The content of the memory location whose address is in the register pair **rp** is copied to register **A**. NOTE: Only register pairs **rp=B** (registers **B** and **C**) or **rp=D** (registers **D** and **E**) may be specified.

Flags: none

**LHLD addr** (Load H and L direct) 0010 1010 ( $2A_{16}$ )

$(L) \leftarrow ((\text{byte } 3)(\text{byte } 2))$

$(H) \leftarrow ((\text{byte } 3)(\text{byte } 2)) + 1$

The content of the memory location whose address is specified in byte 3 and byte 2 of the instruction is copied to register **L**. The content of the memory location at the succeeding address is copied to register **H**.

Flags: none

**LXI rp,data 16** (Load register pair immediate) 00RR 0001 ( $01_{16} + RR*16$ )

$(rh) \leftarrow (\text{byte } 3)$

$(rl) \leftarrow (\text{byte } 2)$

Byte 3 of the instruction is copied into the high-order register (**rh**) of the register pair **rp**. Byte 2 of the instruction is copied into the low-order register (**rh**) of the register pair **rp**.

Flags: none

**MOV M,r** (Move to memory) 0111 0rrr ( $70_{16} + rrr$ )

$((H)(L)) \leftarrow (r)$

The contents of register **r** is moved (copied) to the memory location whose address is in registers **H** and **L**.

Flags: none

**MOV r,M** (Move from memory) 01rr r110 ( $46_{16} + rrr*8$ )

$(r) \leftarrow ((H)(L))$

The contents of the memory location whose address is in registers **H** and **L** is moved (copied) to register **r**.

Flags: none



## A-18 The 8085 Instruction Set

---

**MOV r1,r2** (Move register) 00r1r1 r1r2r2r2 ( $0_{16} + r1r1r1*8 + r2r2r2$ )

$(r1) \leftarrow (r2)$

The contents of register r2 is moved (copied) to register r1.

Flags: none

**MVI M,data** (Move to memory immediate) 0011 0110 ( $36_{16} + rrr$ )

$((H)(L)) \leftarrow (\text{byte } 2)$

The content of byte 2 of the instruction is moved (copied) to the memory location whose address is in registers H and L.

Flags: none

**MVI r,data** (Move to register immediate) 00rr r110 ( $06_{16} + rrr*8$ )

$(r) \leftarrow (\text{byte } 2)$

The content of byte 2 of the instruction is moved (copied) to register r.

Flags: none

**NOP** (No operation) 0000 0000 ( $00_{16}$ )

No operation is performed. The registers and flags are unaffected.

Flags: none

**ORA M** (OR memory) 1011 0110 ( $B6_{16}$ )

$(A) \leftarrow (A) \vee ((H)(L))$

The contents of the memory location whose address is in registers H and L is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**ORA r** (OR register) 1011 0rrr ( $B0_{16} + rrr$ )

$(A) \leftarrow (A) \vee (r)$

The contents of register r is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**ORG addr**                      (Origin statement)                      Assembler Directive

The Origin statement (ORG) sets the initial value of the memory location counter. When the assembler encounters the ORG statement, the memory location counter is set to the address addr. All subsequent object code generated by the assembler is placed in sequential memory locations, starting at the address given by the expression. It is legal to establish a new origin, either before or after a previous origin.

**ORI data**                      (OR Immediate)                      1111 0110 (F6<sub>16</sub>)

$(A) \leftarrow (A) \vee (\text{byte } 2)$

The contents of byte 2 of the instruction is inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**OUT port**                      (Output)                      1101 0011 (D3<sub>16</sub>)

$(\text{date}) \leftarrow (A)$

The content of register A is placed on the eight bit bidirectional data bus for transmission to the specified port.

Flags: none

**PCHL**                      (Jump H and L indirect — move H and L to PC) 1110 1001 (E9<sub>16</sub>)

$(PCH) \leftarrow (H)$

$(PCL) \leftarrow (L)$

The content of register H is copied to the high-order eight bits of register PC. The content of register L is copied to the low-order eight bits of register PC.

Flags: none

**POP rp** (Pop)

11RR 0001 ( $C_{16} + RR \cdot 16$ )

$(r_l) \leftarrow ((SP))$

$(r_h) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

The content of the memory location whose address is specified by the content of register SP is copied to the low-order register of the register pair (rp). The content of the memory location whose address is one more than the content of register SP is copied to the high-order register of register pair rp. The content of register SP is incremented by two. **NOTE: Register pair rp=SP may not be specified.**

Flags: none

**POP PSW** (Pop processor status word)

1111 0001 ( $C_{16}$ )

$(CY) \leftarrow ((SP))_0$

$(P) \leftarrow ((SP))_2$

$(AC) \leftarrow ((SP))_4$

$(Z) \leftarrow ((SP))_6$

$(S) \leftarrow ((SP))_7$

$(A) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

The content of the memory location whose address is specified by the content of register SP is used to restore the condition flags. The content of the memory location whose address is one more than the content of register SP is copied to register A. The content of register SP is incremented by two.

Flags: Z,S,P,CY,AC

11RR 0101 (C5<sub>16</sub> + RR\*16)
$$(\text{SP}) \leftarrow (\text{SP}) - 2$$

Flags: none

1111 0101 (F5<sub>16</sub>)
$$(\text{SP}) \leftarrow (\text{SP}) - 2$$

Flags: none

**RAL** (Rotate left through carry) 0001 0111 (17<sub>16</sub>)

$(A_{n+1}) \leftarrow (A_n)$

$(CY) \leftarrow (A_7)$

$(A_0) \leftarrow (CY)$

The content of the accumulator is rotated left one position through the carry flag. The low-order bit is set equal to the carry flag, and the carry flag is set equal to the value shifted out of the high-order bit position. **Only the CY flag is affected.**

Flags: CY

**RAR** (Rotate right through carry) 0001 1111 (1F<sub>16</sub>)

$(A_n) \leftarrow (A_{n+1})$

$(CY) \leftarrow (A_0)$

$(A_7) \leftarrow (CY)$

The content of the accumulator is rotated right one position through the carry flag. The high-order bit is set equal to the carry flag, and the carry flag is set equal to the value shifted out of the low-order bit position. **Only the CY flag is affected.**

Flags: CY

**RC** (Carry return) 1101 1000 (D8<sub>16</sub>)

If (CY)=1

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

If the carry flag is set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RET** (Return) 1100 1001 ( $C9_{16}$ )

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

The content of the memory location whose address is specified in register SP is copied to the low-order eight bits of register PC. The content of the memory location whose address is one more than the content of register SP is copied to the high-order eight bits of register PC. The content of register SP is incremented by two.

Flags: none

**RIM** (Read interrupt mask) 0010 0000 ( $20_{16}$ )

$(A_0) \leftarrow (M5.5)$

$(A_1) \leftarrow (M6.5)$

$(A_2) \leftarrow (M7.5)$

$(A_3) \leftarrow (IE)$

$(A_4) \leftarrow (I5.5)$

$(A_5) \leftarrow (I6.5)$

$(A_6) \leftarrow (I7.5)$

$(A_7) \leftarrow (SID)$

The interrupt status and serial input data are copied to the accumulator. The interrupt mask bits 0, 1, and 2 (M5.5, M6.5, and M7.5 respectively) are set to 1 when the respective interrupt is masked. Bit 3, IE is set to 1 when the INTR interrupt is enabled. Bits 4, 5, and 6 (I5.5, I6.5, and I7.5 respectively) are set to 1 when the corresponding interrupt has occurred, but not been serviced. Bit 7 receives the value from the serial input data (SID) line.

Flags: none

**RLC** (Rotate left) 0000 0111 (07<sub>16</sub>)

$(A_{n+1}) \leftarrow (A_n)$

$(A_0) \leftarrow (A_7)$

$(CY) \leftarrow (A_7)$

The content of the accumulator is rotated left one position. The low-order bit and the carry flag are both set to the value shifted out of the high-order bit position. **Only the CY flag is affected.**

Flags: CY

**RM** (Minus return) 1111 1000 (F8<sub>16</sub>)

If (S)=1

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

If the sign flag is not set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RNC** (Not carry return) 1101 0000 (D0<sub>16</sub>)

If (CY)=0

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

If the carry flag is not set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RNZ** (Not zero return) 1100 0000 (C0<sub>16</sub>)

If (Z)=0

(PCL)  $\leftarrow$  ((SP))

(PCH)  $\leftarrow$  ((SP) + 1)

(SP)  $\leftarrow$  (SP) + 2

If the zero flag is not set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RP** (Plus return) 1111 0000 (F0<sub>16</sub>)

If (S)=0

(PCL)  $\leftarrow$  ((SP))

(PCH)  $\leftarrow$  ((SP) + 1)

(SP)  $\leftarrow$  (SP) + 2

If the sign flag is set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RPE** (Parity even return) 1110 1000 (E8<sub>16</sub>)

If (P)=1

(PCL)  $\leftarrow$  ((SP))

(PCH)  $\leftarrow$  ((SP) + 1)

(SP)  $\leftarrow$  (SP) + 2

If the parity flag is set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none



**RPO** (Parity odd return) 1110 0000 (E0<sub>16</sub>)

If (P)=0

$(PCL) \leftarrow ((SP))$

$(PCH) \leftarrow ((SP) + 1)$

$(SP) \leftarrow (SP) + 2$

If the parity flag is set, the actions specified in the RET instruction are performed: otherwise, control continues sequentially.

Flags: none

**RRC** (Rotate right) 0000 1111 (0F<sub>16</sub>)

$(A_n) \leftarrow (A_{n+1})$

$(A_7) \leftarrow (A_0)$

$(CY) \leftarrow (A_0)$

The content of the accumulator is rotated right one position. The high-order bit and the carry flag are both set equal to the value shifted out of the low-order bit position. **Only the CY flag is affected.**

Flags: CY

**RST n** (Restart) 00NN N111 (07<sub>16</sub> + NNN\*8)

$((SP) - 1) \leftarrow (PCH)$

$((SP) - 2) \leftarrow (PCL)$

$(SP) \leftarrow (SP) - 2$

$(PC) \leftarrow 8 * NNN$

The high-order eight bits of the next instruction address are moved to the memory location whose address is one less than the content of register SP. The low-order eight bits of the next instruction address are moved to the memory location whose address is two less than the content of register SP. The content of register SP is decremented by two. Control is transferred to the instruction whose address is eight times the value NNN.

Flags: none

**RZ** (Zero return) 1100 1000 (C8<sub>16</sub>)

If (Z)=1

(PCL)  $\leftarrow$  ((SP))

(PCH)  $\leftarrow$  ((SP) + 1)

(SP)  $\leftarrow$  (SP) + 2

If the zero flag is set, the actions specified in the RET instruction are performed; otherwise, control continues sequentially.

Flags: none

**SBB M** (Subtract memory with borrow) 1001 1110 (9E<sub>16</sub>)

(A)  $\leftarrow$  (A) - ((H)(L)) - (CY)

The content of the memory location whose address is contained in the H and L registers and the content of the carry flag are both subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**SBB r** (Subtract register with borrow) 1001 1rrr (AE<sub>16</sub> + rrr)

(A)  $\leftarrow$  (A) - (r) - (CY)

The content of register r and the content of the carry flag are both subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**SBI data** (Subtract immediate with borrow) 1101 1110 (DE<sub>16</sub>)

(A)  $\leftarrow$  (A) - (byte 2) - (CY)

The content of the second byte of the instruction and the content of the carry flag are both subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**SET** (Set statement) Assembler Directive

The SET statement assigns an arbitrary value to a desired symbol. The SET pseudo differs from the EQU pseudo in that any label defined in a SET statement can be redefined in a following SET statement as many times as desired in the course of the program.

**SHLD addr** (Store H and L direct) 0010 0010 (22<sub>16</sub>)

((byte 3)(byte 2)) ← (L)

((byte 3)(byte 2)) + 1 ← (H)

The content of register L is copied to the memory location whose address is specified in byte 3 and byte 2 of the instruction. The content of register H is copied to the memory location at the succeeding address.

Flags: none

**SIM** (Set interrupt mask) 0011 0000 (30<sub>16</sub>)

(M5.5) ← (A<sub>0</sub>)

(M6.5) ← (A<sub>1</sub>)

(M7.5) ← (A<sub>2</sub>)

(MSE) ← (A<sub>3</sub>)

(R7.5) ← (A<sub>4</sub>)

(SOE) ← (A<sub>6</sub>)

(SOD) ← (A<sub>7</sub>)

The interrupt masks, 7.5 reset, and serial output data are copied from the accumulator to their respective destinations. The interrupts are masked by ones in bits 0, 1, and 2 (M5.5, M6.5, and M7.5 respectively). However, no action will be taken unless bit 3 (MSE) is also a one. When Bit 4 is a one, RST 7.5 will be reset (even if it has not been serviced). Serial output enable (SOE) must be set to one to change the level on the serial output data (SOD) line. The new value of SOD will be determined by bit 7.

Flags: none

**SPACE** (Listing control) Assembler Directive

The SPACE pseudo leaves blank lines in the program listing.

**SPHL** (Move HL to SP) 1111 1001 (F9<sub>16</sub>)

(SP) ← (H)(L)

The contents of registers H and L (16 bits) are copied to register SP.

Flags: none

**STA addr** (Store accumulator direct) 0011 0010 (32<sub>16</sub>)

(byte 3)(byte 2) ← (A)

The content of register A is copied to the memory location whose address is specified in byte 3 and byte 2 of the instruction.

Flags: none

**STAX rp** (Store accumulator indirect) 000R 0010 (02<sub>16</sub> + R\*16)

(A) ← ((rp))

The content of register A is copied to the memory location whose address is in the register pair rp. NOTE: Only register pairs rp=B (registers B and C) or rp=D (registers D and E) may be specified.

Flags: none

**STC** (Set the carry) 0011 0111 (37<sub>16</sub>)

(CY) ← 1

The CY flag is set to 1. No other flags are affected.

Flags: CY

**STL 'subtitle'** (Listing control) Assembler Directive

The subtitle pseudo (STL) does not affect pagination. That is, it does not generate a new page, but simply titles a subsection of the program with the string corresponding to 'subtitle'. Subtitles are frequently used to indicate subroutines or major program modules.

**SUB M** (Subtract memory) 1001 0110 ( $96_{16}$ )

$(A) \leftarrow (A) - ((H)(L))$

The content of the memory location whose address is contained in the H and L registers is subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**SUB r** (Subtract register) 1001 0rrr ( $90_{16} + rrr$ )

$(A) \leftarrow (A) - (r)$

The content of register r is subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**SUI data** (Subtract immediate) 1101 0110 ( $D6_{16}$ )

$(A) \leftarrow (A) - (\text{byte } 2)$

The content of the second byte of the instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.

Flags: Z,S,P,CY,AC

**TITLE new title'** (Listing control) Assembler Directive

The title pseudo causes a new page to be used. Unless the assembler is already at the top of a page, a new page of the assembly listing is generated. This page is given the title contained in the string 'new title'.

**XCHG** (Exchange H and L with D and E) 1110 1011 ( $EB_{16}$ )

$(L) \longleftrightarrow (E)$

$(H) \longleftrightarrow (D)$

The content of register L is exchanged with the content of register E. The content of register H is exchanged with the content of register D.

Flags: none

**XRAM** (Exclusive OR memory) 1010 1110 (AE<sub>16</sub>)

$(A) \leftarrow (A) \nabla ((H)(L))$

The contents of the memory location whose address is in registers H and L is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**XRA r** (Exclusive OR register) 1010 1rrr (A8<sub>16</sub> + rrr)

$(A) \leftarrow (A) \nabla (r)$

The contents of register r is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**XRI data** (Exclusive OR immediate) 1110 1110 (EE<sub>16</sub>)

$(A) \leftarrow (A) \nabla (\text{byte } 2)$

The contents of the second byte of the instruction is exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. **The CY and AC flags are cleared.**

Flags: Z,S,P,CY,AC

**XTHL** (Exchange stack top with HL to SP) 1110 0011 (E3<sub>16</sub>)

$(L) \longleftrightarrow ((SP))$

$(H) \longleftrightarrow ((SP) + 1)$

The content of the L register is exchanged with the content of the memory location whose address is specified by the content of register SP. The content of the H register is exchanged with the content of the memory location whose address is one more than the content of register SP.

Flags: none



*Appendix B*

**The 8085 Data Sheet**



## 8085AH/8085AH-2/8085AH-1 8-BIT HMOS MICROPROCESSORS

- Single +5V Power Supply with 10% Voltage Margins
- 3 MHz, 5 MHz and 6 MHz Selections Available
- 20% Lower Power Consumption than 8085A for 3 MHz and 5 MHz
- 1.3  $\mu$ s Instruction Cycle (8085AH); 0.8  $\mu$ s (8085AH-2); 0.67  $\mu$ s (8085AH-1)
- 100% Compatible with 8085A
- 100% Software Compatible with 8080A
- On-Chip Clock Generator (with External Crystal, LC or RC Network)
- On-Chip System Controller; Advanced Cycle Status Information Available for Large System Control
- Four Vectored Interrupt Inputs (One is Non-Maskable) Plus an 8080A-Compatible Interrupt
- Serial In/Serial Out Port
- Decimal, Binary and Double Precision Arithmetic
- Direct Addressing Capability to 64K Bytes of Memory
- Available in EXPRESS
  - Standard Temperature Range
  - Extended Temperature Range

The Intel® 8085AH is a complete 8 bit parallel Central Processing Unit (CPU) implemented in N-channel, depletion load, silicon gate technology (HMOS). Its instruction set is 100% software compatible with the 8080A microprocessor, and it is designed to improve the present 8080A's performance by higher system speed. Its high level of system integration allows a minimum system of three IC's [8085AH (CPU), 8156H (RAM/IO) and 8355/8755A (ROM/PROM/IO)] while maintaining total system expandability. The 8085AH-2 and 8085AH-1 are faster versions of the 8085AH.

The 8085AH incorporates all of the features that the 8224 (clock generator) and 8228 (system controller) provided for the 8080A, thereby offering a high level of system integration.

The 8085AH uses a multiplexed data bus. The address is split between the 8 bit address bus and the 8 bit data bus. The on-chip address latches of 8155H/8156H/8355/8755A memory products allow a direct interface with the 8085AH.

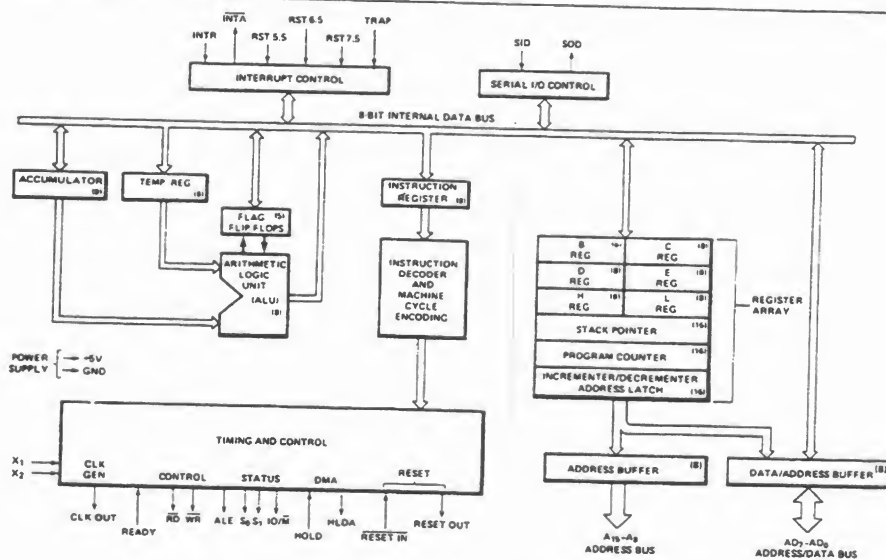


Figure 1. 8085AH CPU Functional Block Diagram

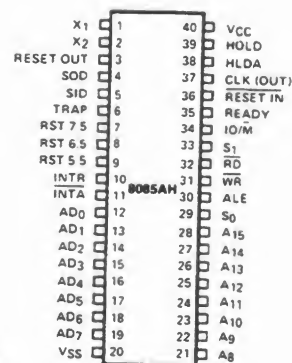


Figure 2. 8085AH Pin Configuration

Table 1. Pin Description

Symbol	Type	Name and Function	Symbol	Type	Name and Function																																												
A <sub>8</sub> -A <sub>15</sub>	O	<b>Address Bus:</b> The most significant 8 bits of the memory address or the 8 bits of the I/O address, 3-stated during Hold and Halt modes and during RESET.	READY	I	<b>Ready:</b> If READY is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If READY is low, the cpu will wait an integral number of clock cycles for READY to go high before completing the read or write cycle. READY must conform to specified setup and hold times.																																												
AD <sub>0</sub> -7	I/O	<b>Multiplexed Address/Data Bus:</b> Lower 8 bits of the memory address (or I/O address) appear on the bus during the first clock cycle (T state) of a machine cycle. It then becomes the data bus during the second and third clock cycles.	HOLD	I	<b>Hold:</b> Indicates that another master is requesting the use of the address and data buses. The cpu, upon receiving the hold request, will relinquish the use of the bus as soon as the completion of the current bus transfer. Internal processing can continue. The processor can regain the bus only after the HOLD is removed. When the HOLD is acknowledged, the Address, Data $\overline{RD}$ , $\overline{WR}$ , and IO/ $\overline{M}$ lines are 3-stated.																																												
ALE	O	<b>Address Latch Enable:</b> It occurs during the first clock state of a machine cycle and enables the address to get latched into the on-chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information. The falling edge of ALE can also be used to strobe the status information. ALE is never 3-stated.	HLDA	O	<b>Hold Acknowledge:</b> Indicates that the cpu has received the HOLD request and that it will relinquish the bus in the next clock cycle. HLDA goes low after the Hold request is removed. The cpu takes the bus one half clock cycle after HLDA goes low.																																												
S <sub>0</sub> , S <sub>1</sub> , and IO/ $\overline{M}$	O	<b>Machine Cycle Status:</b> <table><thead><tr><th>IO/<math>\overline{M}</math></th><th>S<sub>1</sub></th><th>S<sub>0</sub></th><th>Status</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>1</td><td>Memory write</td></tr><tr><td>0</td><td>1</td><td>0</td><td>Memory read</td></tr><tr><td>1</td><td>0</td><td>1</td><td>I/O write</td></tr><tr><td>1</td><td>1</td><td>0</td><td>I/O read</td></tr><tr><td>0</td><td>1</td><td>1</td><td>Opcode fetch</td></tr><tr><td>1</td><td>1</td><td>1</td><td>Opcode fetch</td></tr><tr><td>1</td><td>1</td><td>1</td><td>Interrupt Acknowledge</td></tr><tr><td>*</td><td>0</td><td>0</td><td>Halt</td></tr><tr><td>*</td><td>X</td><td>X</td><td>Hold</td></tr><tr><td>*</td><td>X</td><td>X</td><td>Reset</td></tr></tbody></table> <p>* = 3-state (high impedance) X = unspecified</p> <p>S<sub>1</sub> can be used as an advanced R/<math>\overline{W}</math> status. IO/<math>\overline{M}</math>, S<sub>0</sub> and S<sub>1</sub> become valid at the beginning of a machine cycle and remain stable throughout the cycle. The falling edge of ALE may be used to latch the state of these lines.</p>	IO/ $\overline{M}$	S <sub>1</sub>	S <sub>0</sub>	Status	0	0	1	Memory write	0	1	0	Memory read	1	0	1	I/O write	1	1	0	I/O read	0	1	1	Opcode fetch	1	1	1	Opcode fetch	1	1	1	Interrupt Acknowledge	*	0	0	Halt	*	X	X	Hold	*	X	X	Reset	INTR	I	<b>Interrupt Request:</b> Is used as a general purpose interrupt. It is sampled only during the next to the last clock cycle of an instruction and during Hold and Halt states. If it is active, the Program Counter (PC) will be inhibited from incrementing and an INTA will be issued. During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.
IO/ $\overline{M}$	S <sub>1</sub>	S <sub>0</sub>	Status																																														
0	0	1	Memory write																																														
0	1	0	Memory read																																														
1	0	1	I/O write																																														
1	1	0	I/O read																																														
0	1	1	Opcode fetch																																														
1	1	1	Opcode fetch																																														
1	1	1	Interrupt Acknowledge																																														
*	0	0	Halt																																														
*	X	X	Hold																																														
*	X	X	Reset																																														
$\overline{RD}$	O	<b>Read Control:</b> A low level on $\overline{RD}$ indicates the selected memory or I/O device is to be read and that the Data Bus is available for the data transfer, 3-stated during Hold and Halt modes and during RESET.	INTA	O	<b>Interrupt Acknowledge:</b> Is used instead of (and has the same timing as) $\overline{RD}$ during the Instruction cycle after an INTR is accepted. It can be used to activate an 8259A Interrupt chip or some other interrupt port.																																												
$\overline{WR}$	O	<b>Write Control:</b> A low level on $\overline{WR}$ indicates the data on the Data Bus is to be written into the selected memory or I/O location. Data is set up at the trailing edge of $\overline{WR}$ . 3-stated during Hold and Halt modes and during RESET.	RST 5.5 RST 6.5 RST 7.5	I	<b>Restart Interrupts:</b> These three inputs have the same timing as INTR except they cause an internal RESTART to be automatically inserted.  The priority of these interrupts is ordered as shown in Table 2. These interrupts have a higher priority than INTR. In addition, they may be individually masked out using the SIM instruction.																																												

Table 1. Pin Description (Continued)

Symbol	Type	Name and Function	Symbol	Type	Name and Function
TRAP	I	<b>Trap:</b> Trap interrupt is a non-maskable RESTART interrupt. It is recognized at the same time as INTR or RST 5.5-7.5. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt. (See Table 2.)	RESET OUT	O	<b>Reset Out:</b> Reset Out indicates cpu is being reset. Can be used as a system reset. The signal is synchronized to the processor clock and lasts an integral number of clock periods.
RESET IN	I	<b>Reset In:</b> Sets the Program Counter to zero and resets the Interrupt Enable and HLDA flip-flops. The data and address buses and the control lines are 3-stated during RESET and because of the asynchronous nature of RESET, the processor's internal registers and flags may be altered by RESET with unpredictable results. RESET IN is a Schmitt-triggered input, allowing connection to an R-C network for power-on RESET delay (see Figure 3). Upon power-up, RESET IN must remain low for at least 10 ms after minimum $V_{CC}$ has been reached. For proper reset operation after the power-up duration, RESET IN should be kept low a minimum of three clock periods. The CPU is held in the reset condition as long as RESET IN is applied.	$X_1, X_2$	I	<b><math>X_1</math> and <math>X_2</math>:</b> Are connected to a crystal, LC, or RC network to drive the internal clock generator. $X_1$ can also be an external clock input from a logic gate. The input frequency is divided by 2 to give the processor's internal operating frequency.
			CLK	O	<b>Clock:</b> Clock output for use as a system clock. The period of CLK is twice the $X_1, X_2$ input period.
			SID	I	<b>Serial Input Data Line:</b> The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.
			SOD	O	<b>Serial Output Data Line:</b> The output SOD is set or reset as specified by the SIM instruction.
			$V_{CC}$		<b>Power:</b> +5 volt supply.
			$V_{SS}$		<b>Ground:</b> Reference.

Table 2. Interrupt Priority, Restart Address, and Sensitivity

Name	Priority	Address Branched To (1) When Interrupt Occurs	Type Trigger
TRAP	1	24H	Rising edge AND high level until sampled.
RST 7.5	2	3CH	Rising edge (latched).
RST 6.5	3	34H	High level until sampled.
RST 5.5	4	2CH	High level until sampled.
INTR	5	See Note (2).	High level until sampled.

**NOTES:**

1. The processor pushes the PC on the stack before branching to the indicated address.
2. The address branched to depends on the instruction provided to the cpu when the interrupt is acknowledged.

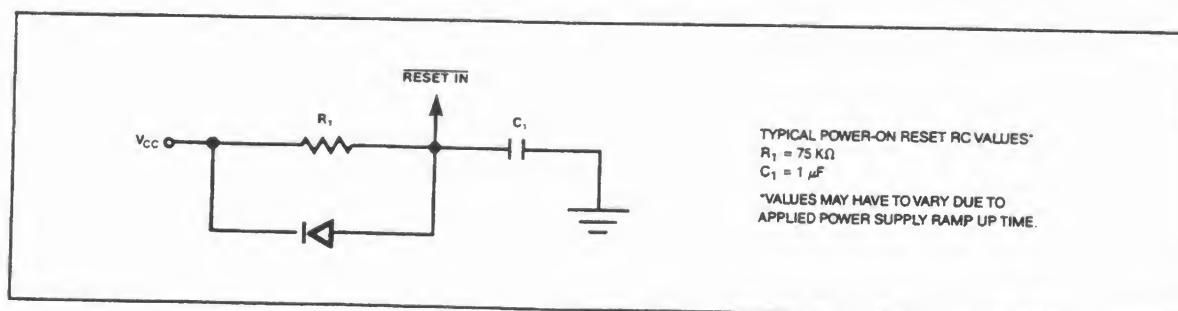


Figure 3. Power-On Reset Circuit

## FUNCTIONAL DESCRIPTION

The 8085AH is a complete 8-bit parallel central processor. It is designed with N-channel, depletion load, silicon gate technology (HMOS), and requires a single +5 volt supply. Its basic clock speed is 3 MHz (8085AH), 5 MHz (8085AH-2), or 6 MHz (8085AH-1), thus improving on the present 8080A's performance with higher system speed. Also it is designed to fit into a minimum system of three IC's: The CPU (8085AH), a RAM/IO (8156H), and a ROM or EPROM/IO chip (8355 or 8755A).

The 8085AH has twelve addressable 8-bit registers. Four of them can function only as two 16-bit register pairs. Six others can be used interchangeably as 8-bit registers or as 16-bit register pairs. The 8085AH register set is as follows:

Mnemonic	Register	Contents
ACC or A	Accumulator	8 bits
PC	Program Counter	16-bit address
BC,DE,HL	General-Purpose Registers; data pointer (HL)	8 bits x 6 or 16 bits x 3
SP	Stack Pointer	16-bit address
Flags or F	Flag Register	5 flags (8-bit space)

The 8085AH uses a multiplexed Data Bus. The address is split between the higher 8-bit Address Bus and the lower 8-bit Address/Data Bus. During the first T state (clock cycle) of a machine cycle the low order address is sent out on the Address/Data bus. These lower 8 bits may be latched externally by the Address Latch Enable signal (ALE). During the rest of the machine cycle the data bus is used for memory or I/O data.

The 8085AH provides  $\overline{RD}$ ,  $\overline{WR}$ ,  $S_0$ ,  $S_1$ , and  $IO/\overline{M}$  signals for bus control. An Interrupt Acknowledge signal ( $\overline{INTA}$ ) is also provided. HOLD and all Interrupts are synchronized with the processor's internal clock. The 8085AH also provides Serial Input Data (SID) and Serial Output Data (SOD) lines for simple serial interface.

In addition to these features, the 8085AH has three maskable, vector interrupt pins, one nonmaskable TRAP interrupt, and a bus vectored interrupt, INTR.

## INTERRUPT AND SERIAL I/O

The 8085AH has 5 interrupt inputs: INTR, RST 5.5, RST 6.5, RST 7.5, and TRAP. INTR is identical in function to the 8080A INT. Each of the three RESTART inputs, 5.5, 6.5, and 7.5, has a programmable mask. TRAP is also a RESTART interrupt but it is nonmaskable.

The three maskable interrupts cause the internal execution of RESTART (saving the program counter in the stack and branching to the RESTART address) if the interrupts are enabled and if the interrupt mask is not set. The nonmaskable TRAP causes the internal execution of a RESTART vector independent of the state of the interrupt enable or masks. (See Table 2.)

There are two different types of inputs in the restart interrupts. RST 5.5 and RST 6.5 are *high level-sensitive* like INTR (and INT on the 8080) and are recognized with the same timing as INTR. RST 7.5 is *rising edge-sensitive*.

For RST 7.5, only a pulse is required to set an internal flip-flop which generates the internal interrupt request (a normally high level signal with a low going pulse is recommended for highest system noise immunity). The RST 7.5 request flip-flop remains set until the request is serviced. Then it is reset automatically. This flip-flop may also be reset by using the SIM instruction or by issuing a  $\overline{RESET IN}$  to the 8085AH. The RST 7.5 internal flip-flop will be set by a pulse on the RST 7.5 pin even when the RST 7.5 interrupt is masked out.

The status of the three RST interrupt masks can only be affected by the SIM instruction and  $\overline{RESET IN}$ . (See SIM, Chapter 5 of the MCS-80/85 User's Manual.)

The interrupts are arranged in a fixed priority that determines which interrupt is to be recognized if more than one is pending as follows: TRAP—highest priority, RST 7.5, RST 6.5, RST 5.5, INTR—lowest priority. This priority scheme does not take into account the priority of a routine that was started by a higher priority interrupt. RST 5.5 can interrupt an RST 7.5 routine if the interrupts are re-enabled before the end of the RST 7.5 routine.

The TRAP interrupt is useful for catastrophic events such as power failure or bus error. The TRAP input is recognized just as any other interrupt but has the highest priority. It is not affected by any flag or mask. The TRAP input is both *edge and level sensitive*. The TRAP input must go high and remain high until it is acknowledged. It will not be recognized again until it goes low, then high again. This avoids any false triggering due to noise or logic glitches. Figure 4 illustrates the TRAP interrupt request circuitry within the 8085AH. Note that the servicing of any interrupt (TRAP, RST 7.5, RST 6.5, RST 5.5, INTR) disables all future interrupts (except TRAPs) until an EI instruction is executed.

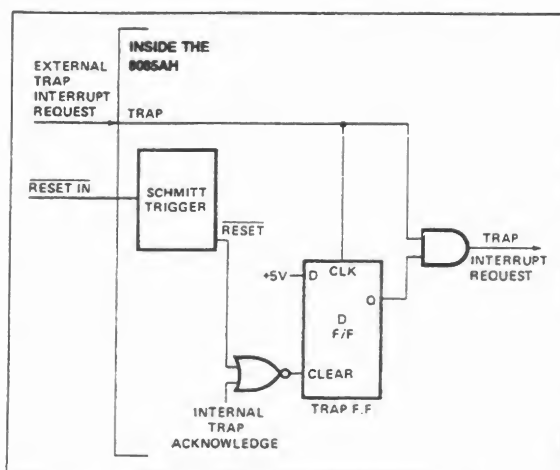


Figure 4. TRAP and RESET IN Circuit

The TRAP interrupt is special in that it disables interrupts, but preserves the previous interrupt enable status. Performing the first RIM instruction following a TRAP interrupt allows you to determine whether interrupts were enabled or disabled prior to the TRAP. All subsequent RIM instructions provide current interrupt enable status. Performing a RIM instruction following INTR, or RST 5.5-7.5 will provide current Interrupt Enable status, revealing that Interrupts are disabled. See the description of the RIM instruction in the MCS-80/85 Family User's Manual.

The serial I/O system is also controlled by the RIM and SIM instructions. SID is read by RIM, and SIM sets the SOD data.

### DRIVING THE X<sub>1</sub> AND X<sub>2</sub> INPUTS

You may drive the clock inputs of the 8085AH, 8085AH-2, or 8085AH-1 with a crystal, an LC tuned circuit, an RC network, or an external clock source. The crystal frequency must be at least 1 MHz, and must be twice the desired internal clock frequency; hence, the 8085AH is operated with a 6 MHz crystal (for 3 MHz clock), the 8085AH-2 operated with a 10 MHz crystal (for 5 MHz clock), and the 8085AH-1 can be operated with a 12 MHz crystal (for 6 MHz clock). If a crystal is used, it must have the following characteristics:

Parallel resonance at twice the clock frequency desired

$C_L$  (load capacitance)  $\leq 30$  pF

$C_S$  (shunt capacitance)  $\leq 7$  pF

$R_S$  (equivalent shunt resistance)  $\leq 75$  Ohms

Drive level: 10 mW

Frequency tolerance:  $\pm .005\%$  (suggested)

Note the use of the 20 pF capacitor between X<sub>2</sub> and ground. This capacitor is required with crystal frequencies below 4 MHz to assure oscillator startup at the correct frequency. A parallel-resonant LC circuit may be used as the frequency-determining network for the 8085AH, providing that its frequency tolerance of approximately  $\pm 10\%$  is acceptable. The components are chosen from the formula:

$$f = \frac{1}{2\pi\sqrt{L(C_{\text{ext}} + C_{\text{int}})}}$$

To minimize variations in frequency, it is recommended that you choose a value for  $C_{\text{ext}}$  that is at least twice that of  $C_{\text{int}}$ , or 30 pF. The use of an LC circuit is not recommended for frequencies higher than approximately 5 MHz.

An RC circuit may be used as the frequency-determining network for the 8085AH if maintaining a precise clock frequency is of no importance. Variations in the on-chip timing generation can cause a wide variation in frequency when using the RC mode. Its advantage is its low component cost. The driving frequency generated by the circuit shown is approximately 3 MHz. It is not recommended that frequencies greatly higher or lower than this be attempted.

Figure 5 shows the recommended clock driver circuits. Note in D and E that pullup resistors are required to assure that the high level voltage of the input is at least 4V and maximum low level voltage of 0.8V.

For driving frequencies up to and including 6 MHz you may supply the driving signal to X<sub>1</sub> and leave X<sub>2</sub> open-circuited (Figure 5D). If the driving frequency is from 6 MHz to 12 MHz, stability of the clock generator will be improved by driving both X<sub>1</sub> and X<sub>2</sub> with a push-pull source (Figure 5E). To prevent self-oscillation of the 8085AH, be sure that X<sub>2</sub> is not coupled back to X<sub>1</sub> through the driving circuit.

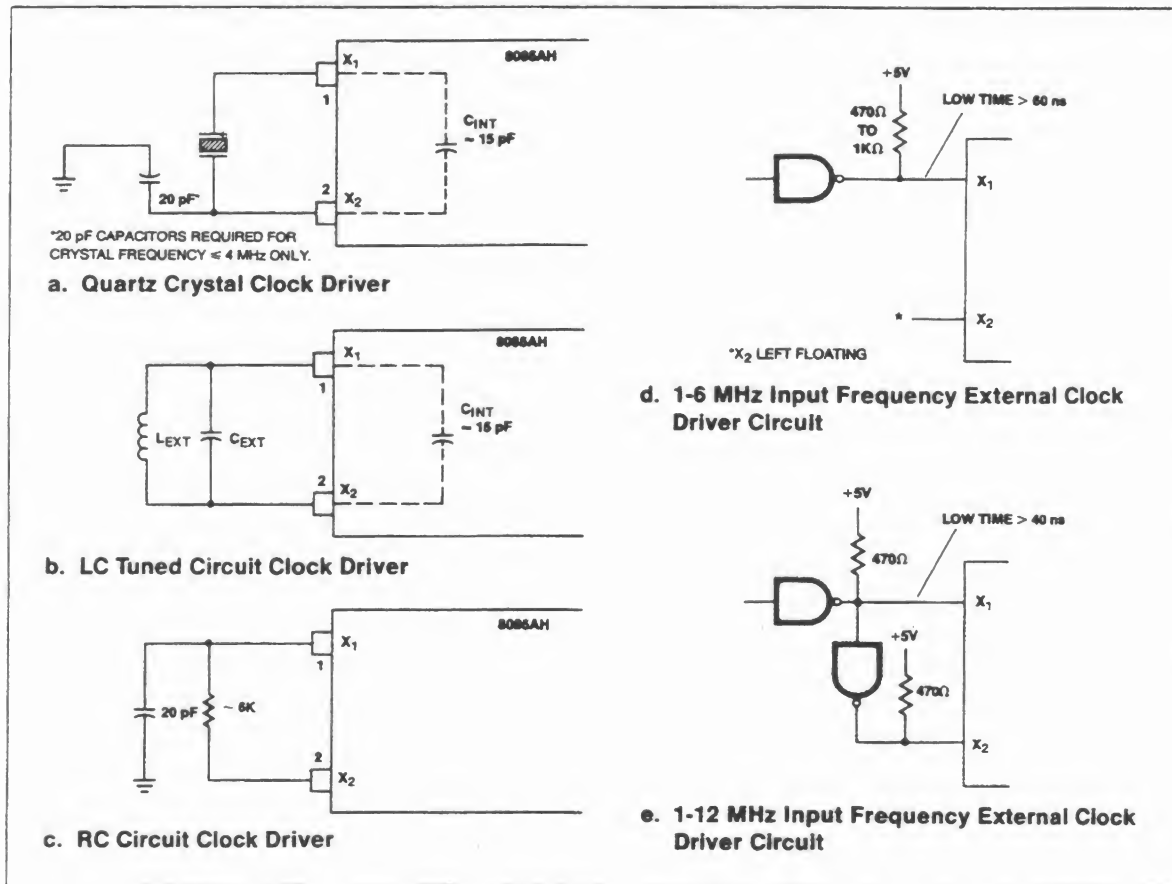


Figure 5. Clock Driver Circuits

### GENERATING AN 8085AH WAIT STATE

If your system requirements are such that slow memories or peripheral devices are being used, the circuit shown in Figure 6 may be used to insert one WAIT state in each 8085AH machine cycle.

The D flip-flops should be chosen so that

- CLK is rising edge-triggered
- CLEAR is low-level active.

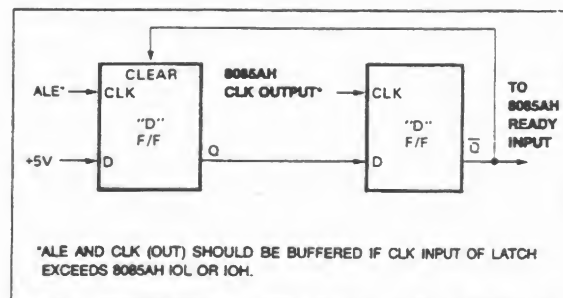
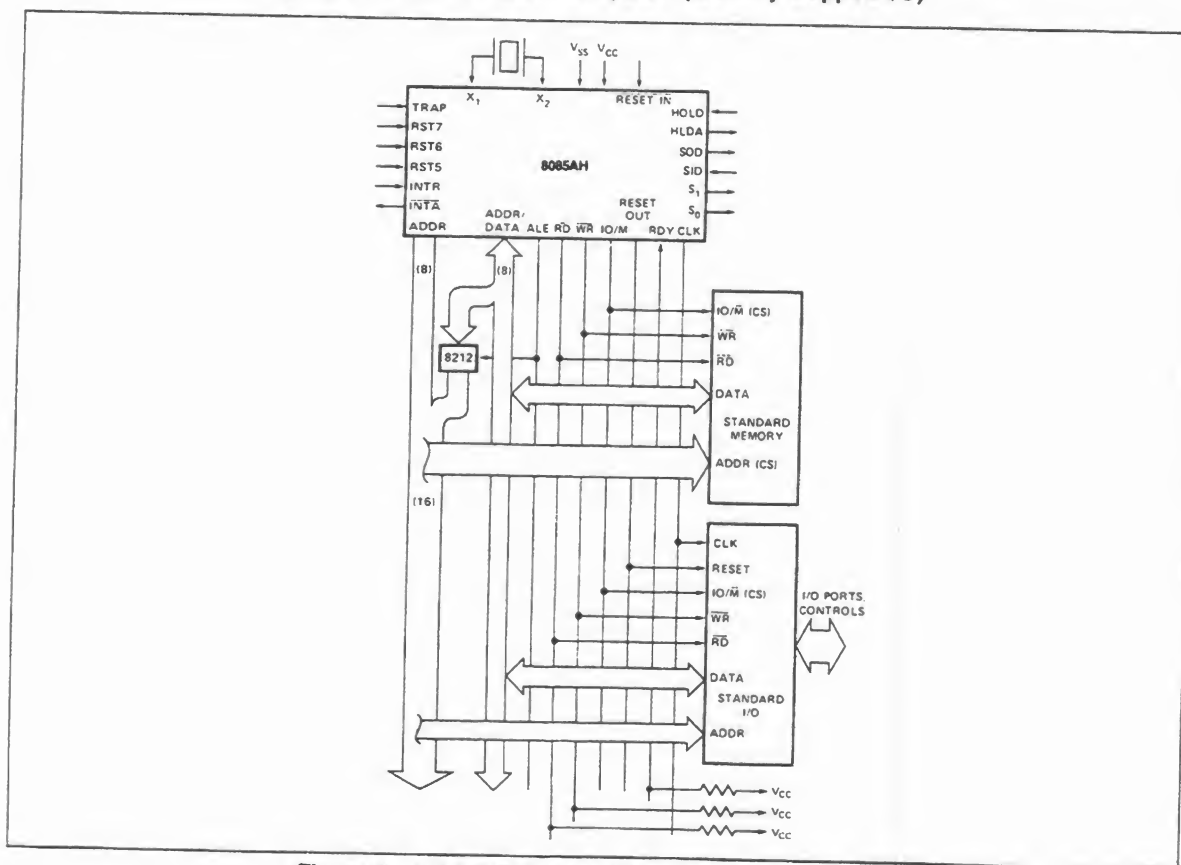
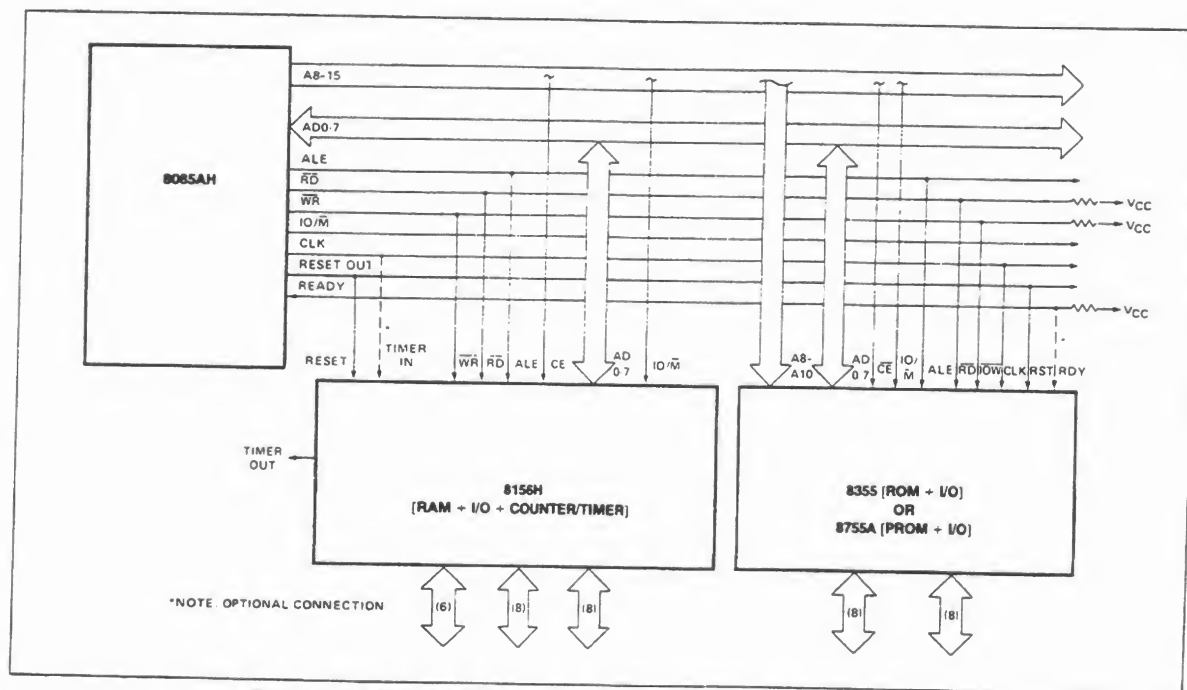


Figure 6. Generation of a Wait State for 8085AH CPU





As in the 8080, the READY line is used to extend the read and write pulse lengths so that the 8085AH can be used with slow memory. HOLD causes the CPU to relinquish the bus when it is through with it by floating the Address and Data Buses.

## SYSTEM INTERFACE

The 8085AH family includes memory components, which are directly compatible to the 8085AH CPU. For example, a system consisting of the three chips, 8085AH, 8156H, and 8355 will have the following features:

- 2K Bytes ROM
- 256 Bytes RAM
- 1 Timer/Counter
- 4 8-bit I/O Ports
- 1 6-bit I/O Port
- 4 Interrupt Levels
- Serial In/Serial Out Ports

This minimum system, using the standard I/O technique is as shown in Figure 7.

In addition to standard I/O, the memory mapped I/O offers an efficient I/O addressing technique. With this technique, an area of memory address space is assigned for I/O address, thereby, using the memory address for I/O manipulation. Figure 8 shows the system configuration of Memory Mapped I/O using 8085AH.

The 8085AH CPU can also interface with the standard memory that does *not* have the multiplexed address/data bus. It will require a simple 8212 (8-bit latch) as shown in Figure 9.

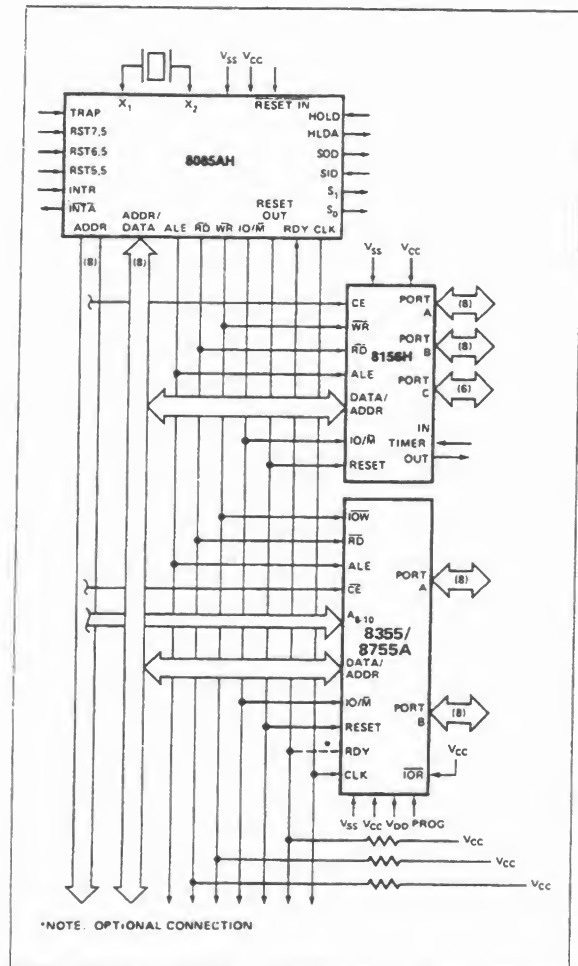


Figure 7. 8085AH Minimum System (Standard I/O Technique)



## BASIC SYSTEM TIMING

The 8085AH has a multiplexed Data Bus. ALE is used as a strobe to sample the lower 8-bits of address on the Data Bus. Figure 10 shows an instruction fetch, memory read and I/O write cycle (as would occur during processing of the OUT instruction). Note that during the I/O write and read cycle that the I/O port address is copied on both the upper and lower half of the address.

There are seven possible types of machine cycles. Which of these seven takes place is defined by the status of the three status lines ( $\overline{IO/\overline{M}}$ ,  $S_1$ ,  $S_0$ ) and the three control signals ( $\overline{RD}$ ,  $\overline{WR}$ , and  $\overline{INTA}$ ). (See Table 3.) The status lines can be used as advanced controls (for device selection, for example), since they become active at the  $T_1$  state, at the outset of each machine cycle. Control lines  $\overline{RD}$  and  $\overline{WR}$  become active later, at the time when the transfer of data is to take place, so are used as command lines.

A machine cycle normally consists of three T states, with the exception of OPCODE FETCH, which normally has either four or six T states (unless WAIT or HOLD states are forced by the receipt of  $\overline{READY}$  or  $\overline{HOLD}$  inputs). Any T state must be one of ten possible states, shown in Table 4.

Table 3. 8085AH Machine Cycle Chart

MACHINE CYCLE		STATUS			CONTROL		
		$\overline{IO/\overline{M}}$	$S_1$	$S_0$	$\overline{RD}$	$\overline{WR}$	$\overline{INTA}$
OPCODE FETCH (OF)		0	1	1	0	1	1
MEMORY READ (MR)		0	1	0	0	1	1
MEMORY WRITE (MW)		0	0	1	1	0	1
I/O READ (IOR)		1	1	0	0	1	1
I/O WRITE (IOW)		1	0	1	1	0	1
ACKNOWLEDGE OF INTR (INA)		1	1	1	1	1	0
BUS IDLE (BI): DAD		0	1	0	1	1	1
ACK. OF RST, TRAP		1	1	1	1	1	1
HALT		TS	0	0	TS	TS	1

Table 4. 8085AH Machine State Chart

Machine State	Status & Buses				Control		
	$S_1, S_0$	$\overline{IO/\overline{M}}$	$A_8-A_{15}$	$AD_0-AD_7$	$\overline{RD}, \overline{WR}$	$\overline{INTA}$	ALE
$T_1$	X	X	X	X	1	1	1*
$T_2$	X	X	X	X	X	X	0
$T_{WAIT}$	X	X	X	X	X	X	0
$T_3$	X	X	X	X	X	X	0
$T_4$	1	0†	X	TS	1	1	0
$T_5$	1	0†	X	TS	1	1	0
$T_6$	1	0†	X	TS	1	1	0
$T_{RESET}$	X	TS	TS	TS	TS	1	0
$T_{HALT}$	0	TS	TS	TS	TS	1	0
$T_{HOLD}$	X	TS	TS	TS	TS	1	0

0 = Logic "0"

1 = Logic "1"

TS = High Impedance

X = Unspecified

\* ALE not generated during 2nd and 3rd machine cycles of DAD instruction.

†  $\overline{IO/\overline{M}} = 1$  during  $T_4-T_6$  of INA machine cycle.

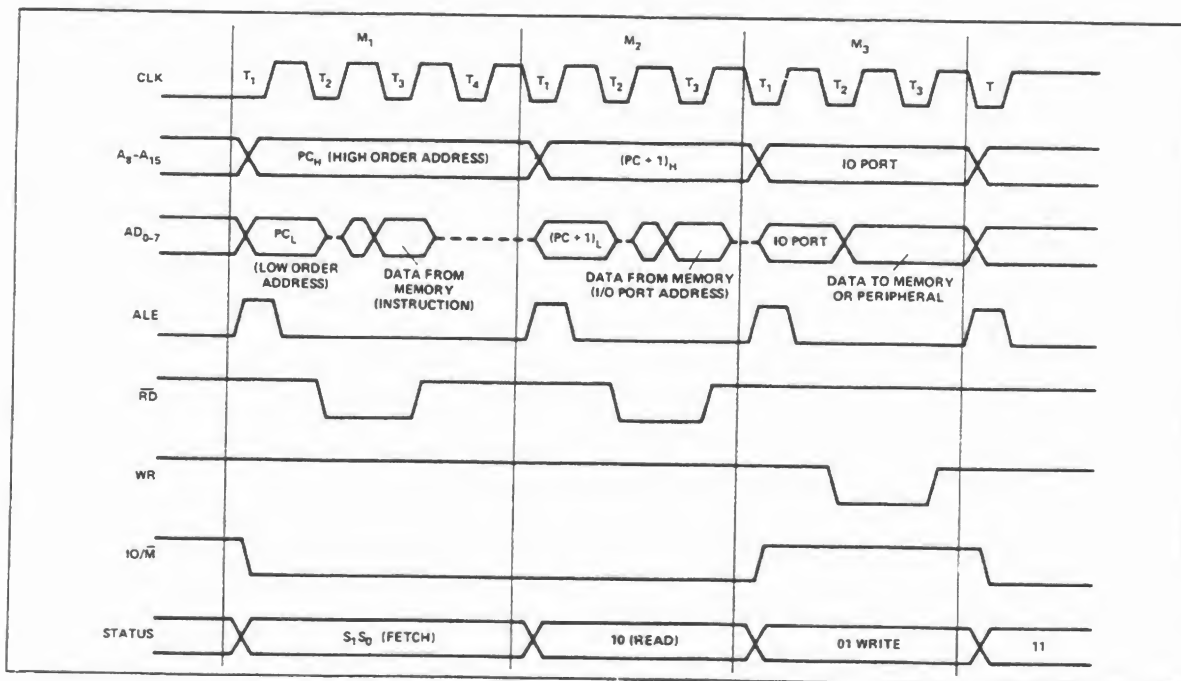


Figure 10. 8085AH Basic System Timing

**ABSOLUTE MAXIMUM RATINGS\***

Ambient Temperature Under Bias ..... 0°C to 70°C  
 Storage Temperature ..... -65°C to +150°C  
 Voltage on Any Pin  
   With Respect to Ground ..... -0.5V to +7V  
 Power Dissipation ..... 1.5 Watt

*\*NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

**D.C. CHARACTERISTICS**

8085AH, 8085AH-2: ( $T_A = 0^\circ\text{C}$  to  $70^\circ\text{C}$ ,  $V_{CC} = 5V \pm 10\%$ ,  $V_{SS} = 0V$ ; unless otherwise specified)\*  
 8085AH-1: ( $T_A = 0^\circ\text{C}$  to  $70^\circ\text{C}$ ,  $V_{CC} = 5V \pm 5\%$ ,  $V_{SS} = 0V$ ; unless otherwise specified)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
$V_{IL}$	Input Low Voltage	-0.5	+0.8	V	
$V_{IH}$	Input High Voltage	2.0	$V_{CC} + 0.5$	V	
$V_{OL}$	Output Low Voltage		0.45	V	$I_{OL} = 2\text{mA}$
$V_{OH}$	Output High Voltage	2.4		V	$I_{OH} = -400\mu\text{A}$
$I_{CC}$	Power Supply Current		135	mA	8085AH, 8085AH-2
			200	mA	8085AH-1 (Preliminary)
$I_{IL}$	Input Leakage		$\pm 10$	$\mu\text{A}$	$0 \leq V_{IN} \leq V_{CC}$
$I_{LO}$	Output Leakage		$\pm 10$	$\mu\text{A}$	$0.45V \leq V_{OUT} \leq V_{CC}$
$V_{ILR}$	Input Low Level, RESET	-0.5	+0.8	V	
$V_{IHR}$	Input High Level, RESET	2.4	$V_{CC} + 0.5$	V	
$V_{HY}$	Hysteresis, RESET	0.25		V	

**A.C. CHARACTERISTICS**

8085AH, 8085AH-2: ( $T_A = 0^\circ\text{C}$  to  $70^\circ\text{C}$ ,  $V_{CC} = 5V \pm 10\%$ ,  $V_{SS} = 0V$ )\*  
 8085AH-1: ( $T_A = 0^\circ\text{C}$  to  $70^\circ\text{C}$ ,  $V_{CC} = 5V \pm 5\%$ ,  $V_{SS} = 0V$ )

Symbol	Parameter	8085AH <sup>[2]</sup> (Final)		8085AH-2 <sup>[2]</sup> (Final)		8085AH-1 (Preliminary)		Units
		Min.	Max.	Min.	Max.	Min.	Max.	
$t_{CYC}$	CLK Cycle Period	320	2000	200	2000	167	2000	ns
$t_1$	CLK Low Time (Standard CLK Loading)	80		40		20		ns
$t_2$	CLK High Time (Standard CLK Loading)	120		70		50		ns
$t_r, t_f$	CLK Rise and Fall Time		30		30		30	ns
$t_{XKR}$	$X_1$ Rising to CLK Rising	25	120	25	100	20	100	ns
$t_{XKF}$	$X_1$ Rising to CLK Falling	30	150	30	110	25	110	ns
$t_{AC}$	$A_{8-15}$ Valid to Leading Edge of Control <sup>[1]</sup>	270		115		70		ns
$t_{ACL}$	$A_{0-7}$ Valid to Leading Edge of Control	240		115		60		ns
$t_{AD}$	$A_{0-15}$ Valid to Valid Data In		575		350		225	ns
$t_{AFR}$	Address Float After Leading Edge of READ (INTA)		0		0		0	ns
$t_{AL}$	$A_{8-15}$ Valid Before Trailing Edge of ALE <sup>[1]</sup>	115		50		25		ns

\*Note: For Extended Temperature EXPRESS use M8085AH Electricals Parameters.

**A.C. CHARACTERISTICS (Continued)**

Symbol	Parameter	8085AH <sup>[2]</sup> (Final)		8085AH-2 <sup>[2]</sup> (Final)		8085AH-1 (Preliminary)		Units
		Min.	Max.	Min.	Max.	Min.	Max.	
t <sub>ALL</sub>	A <sub>0-7</sub> Valid Before Trailing Edge of ALE	90		50		25		ns
t <sub>ARY</sub>	READY Valid from Address Valid		220		100		40	ns
t <sub>CA</sub>	Address (A <sub>8-15</sub> ) Valid After Control	120		60		30		ns
t <sub>CC</sub>	Width of Control Low ( $\overline{RD}$ , $\overline{WR}$ , $\overline{INTA}$ ) Edge of ALE	400		230		150		ns
t <sub>CL</sub>	Trailing Edge of Control to Leading Edge of ALE	50		25		0		ns
t <sub>DW</sub>	Data Valid to Trailing Edge of $\overline{WRITE}$	420		230		140		ns
t <sub>HABE</sub>	HLDA to Bus Enable		210		150		150	ns
t <sub>HABF</sub>	Bus Float After HLDA		210		150		150	ns
t <sub>HACK</sub>	HLDA Valid to Trailing Edge of CLK	110		40		0		ns
t <sub>HDH</sub>	HOLD Hold Time	0		0		0		ns
t <sub>HDS</sub>	HOLD Setup Time to Trailing Edge of CLK	170		120		120		ns
t <sub>INH</sub>	INTR Hold Time	0		0		0		ns
t <sub>INS</sub>	INTR, RST, and TRAP Setup Time to Falling Edge of CLK	160		150		150		ns
t <sub>LA</sub>	Address Hold Time After ALE	100		50		20		ns
t <sub>LC</sub>	Trailing Edge of ALE to Leading Edge of Control	130		60		25		ns
t <sub>LCK</sub>	ALE Low During CLK High	100		50		15		ns
t <sub>LDR</sub>	ALE to Valid Data During Read		460		270		175	ns
t <sub>LDW</sub>	ALE to Valid Data During Write		200		120		110	ns
t <sub>LL</sub>	ALE Width	140		80		50		ns
t <sub>LRY</sub>	ALE to READY Stable		110		30		10	ns
t <sub>RAE</sub>	Trailing Edge of $\overline{READ}$ to Re-Enabling of Address	150		90		50		ns
t <sub>RD</sub>	$\overline{READ}$ (or $\overline{INTA}$ ) to Valid Data		300		150		75	ns
t <sub>RV</sub>	Control Trailing Edge to Leading Edge of Next Control	400		220		160		ns
t <sub>RDH</sub>	Data Hold Time After $\overline{READ}$ $\overline{INTA}$	0		0		0		ns
t <sub>RYH</sub>	READY Hold Time	0		0		5		ns
t <sub>RYs</sub>	READY Setup Time to Leading Edge of CLK	110		100		100		ns
t <sub>WD</sub>	Data Valid After Trailing Edge of $\overline{WRITE}$	100		60		30		ns
t <sub>WDL</sub>	LEADING Edge of $\overline{WRITE}$ to Data Valid		40		20		30	ns

**NOTES:**

1.  $A_8-A_{15}$  address Specs apply  $IO/\overline{M}$ ,  $S_0$ , and  $S_1$  except  $A_8-A_{15}$  are undefined during  $T_4-T_6$  of OF cycle whereas  $IO/\overline{M}$ ,  $S_0$ , and  $S_1$  are stable.
2. Test Conditions:  $t_{CYC} = 320$  ns (8085AH)/200 ns (8085AH-2)/167 ns (8085AH-1);  $C_L = 150$  pF.

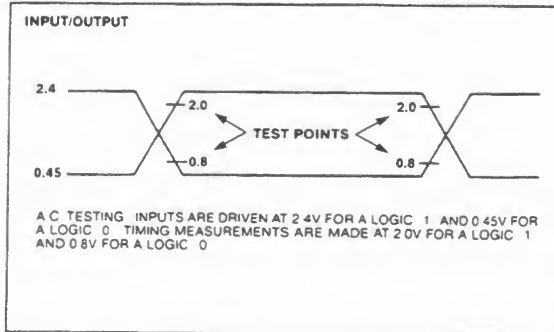
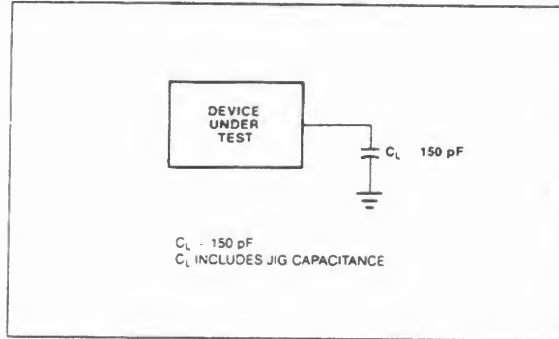
3. For all output timing where  $C_L \neq 150$  pF use the following correction factors:

$$25 \text{ pF} \leq C_L < 150 \text{ pF}: -0.10 \text{ ns/pF}$$

$$150 \text{ pF} < C_L \leq 300 \text{ pF}: +0.30 \text{ ns/pF}$$

4. Output timings are measured with purely capacitive load.

5. To calculate timing specifications at other values of  $t_{CYC}$  use Table 5.

**A.C. TESTING INPUT, OUTPUT WAVEFORM****A.C. TESTING LOAD CIRCUIT****Table 5. Bus Timing Specification as a  $T_{CYC}$  Dependent**

Symbol	8085AH	8085AH-2	8085AH-1	
$t_{AL}$	$(1/2) T - 45$	$(1/2) T - 50$	$(1/2) T - 58$	Minimum
$t_{LA}$	$(1/2) T - 60$	$(1/2) T - 50$	$(1/2) T - 63$	Minimum
$t_{LL}$	$(1/2) T - 20$	$(1/2) T - 20$	$(1/2) T - 33$	Minimum
$t_{LCK}$	$(1/2) T - 60$	$(1/2) T - 50$	$(1/2) T - 68$	Minimum
$t_{LC}$	$(1/2) T - 30$	$(1/2) T - 40$	$(1/2) T - 58$	Minimum
$t_{AD}$	$(5/2 + N) T - 225$	$(5/2 + N) T - 150$	$(5/2 + N) T - 192$	Maximum
$t_{RD}$	$(3/2 + N) T - 180$	$(3/2 + N) T - 150$	$(3/2 + N) T - 175$	Maximum
$t_{RAE}$	$(1/2) T - 10$	$(1/2) T - 10$	$(1/2) T - 33$	Minimum
$t_{CA}$	$(1/2) T - 40$	$(1/2) T - 40$	$(1/2) T - 53$	Minimum
$t_{DW}$	$(3/2 + N) T - 60$	$(3/2 + N) T - 70$	$(3/2 + N) T - 110$	Minimum
$t_{WD}$	$(1/2) T - 60$	$(1/2) T - 40$	$(1/2) T - 53$	Minimum
$t_{CC}$	$(3/2 + N) T - 80$	$(3/2 + N) T - 70$	$(3/2 + N) T - 100$	Minimum
$t_{CL}$	$(1/2) T - 110$	$(1/2) T - 75$	$(1/2) T - 83$	Minimum
$t_{ARY}$	$(3/2) T - 260$	$(3/2) T - 200$	$(3/2) T - 210$	Maximum
$t_{HACK}$	$(1/2) T - 50$	$(1/2) T - 60$	$(1/2) T - 83$	Minimum
$t_{HABF}$	$(1/2) T + 50$	$(1/2) T + 50$	$(1/2) T + 67$	Maximum
$t_{HABE}$	$(1/2) T + 50$	$(1/2) T + 50$	$(1/2) T + 67$	Maximum
$t_{AC}$	$(2/2) T - 50$	$(2/2) T - 85$	$(2/2) T - 97$	Minimum
$t_1$	$(1/2) T - 80$	$(1/2) T - 60$	$(1/2) T - 63$	Minimum
$t_2$	$(1/2) T - 40$	$(1/2) T - 30$	$(1/2) T - 33$	Minimum
$t_{RV}$	$(3/2) T - 80$	$(3/2) T - 80$	$(3/2) T - 90$	Minimum
$t_{LDR}$	$(4/2) T - 180$	$(4/2) T - 130$	$(4/2) T - 159$	Maximum

**NOTE:** N is equal to the total WAIT states.  $T = t_{CYC}$ .

# WAVEFORMS (Continued)

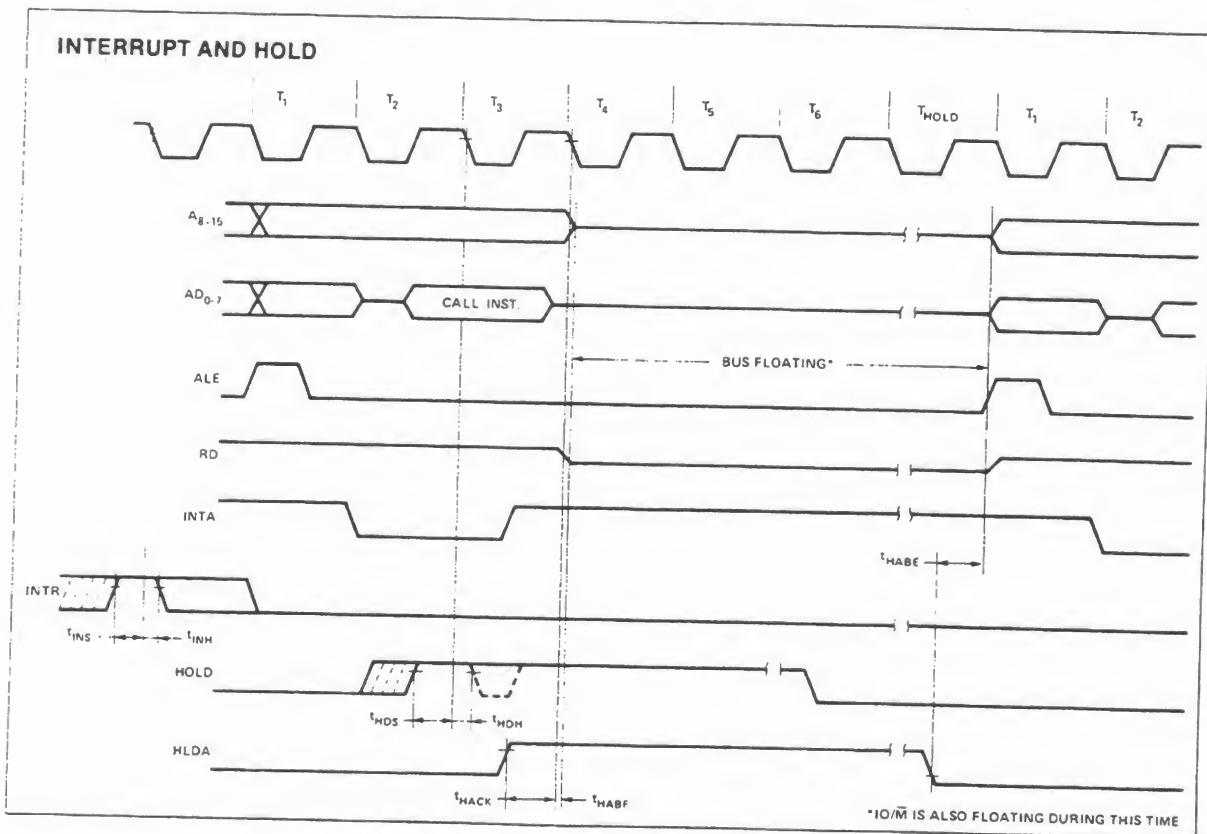
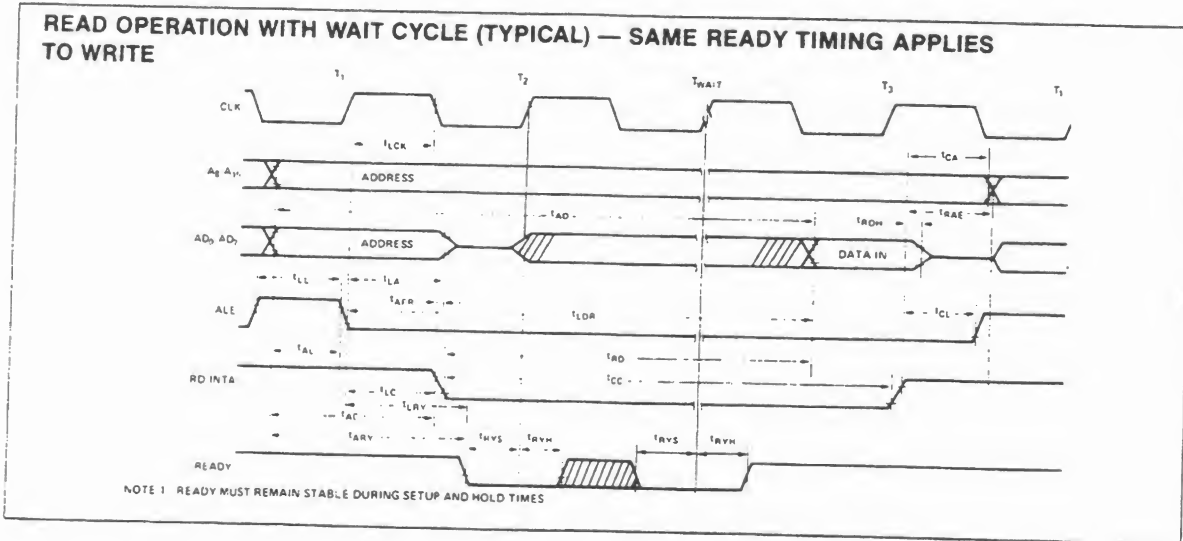


Table 6. Instruction Set Summary

Mnemonic	Instruction Code D <sub>7</sub> D <sub>6</sub> D <sub>5</sub> D <sub>4</sub> D <sub>3</sub> D <sub>2</sub> D <sub>1</sub> D <sub>0</sub>	Operations Description
MOVE, LOAD, AND STORE		
MOV r1 r2	0 1 D D D S S S	Move register to register
MOV M, r	0 1 1 1 0 S S S	Move register to memory
MOV r, M	0 1 D D D 1 1 0	Move memory to register
MVI r	0 0 D D D 1 1 0	Move immediate register
MVI M	0 0 1 1 0 1 1 0	Move immediate memory
LXI B	0 0 0 0 0 0 0 1	Load immediate register Pair B & C
LXI D	0 0 0 1 0 0 0 1	Load immediate register Pair D & E
LXI H	0 0 1 0 0 0 0 1	Load immediate register Pair H & L
STAX B	0 0 0 0 0 0 1 0	Store A indirect
STAX D	0 0 0 1 0 0 1 0	Store A indirect
LDAX B	0 0 0 0 1 0 1 0	Load A indirect
LDAX D	0 0 0 1 1 0 1 0	Load A indirect
STA	0 0 1 1 0 0 1 0	Store A direct
LDA	0 0 1 1 1 0 1 0	Load A direct
SHLD	0 0 1 0 0 0 1 0	Store H & L direct
LHLD	0 0 1 0 1 0 1 0	Load H & L direct
XCHG	1 1 1 0 1 0 1 1	Exchange D & E, H & L Registers
STACK OPS		
PUSH B	1 1 0 0 0 1 0 1	Push register Pair B & C on stack
PUSH D	1 1 0 1 0 1 0 1	Push register Pair D & E on stack
PUSH H	1 1 1 0 0 1 0 1	Push register Pair H & L on stack
PUSH PSW	1 1 1 1 0 1 0 1	Push A and Flags on stack
POP B	1 1 0 0 0 0 0 1	Pop register Pair B & C off stack
POP D	1 1 0 1 0 0 0 1	Pop register Pair D & E off stack
POP H	1 1 1 0 0 0 0 1	Pop register Pair H & L off stack
POP PSW	1 1 1 1 0 0 0 1	Pop A and Flags off stack
XTHL	1 1 1 0 0 0 1 1	Exchange top of stack, H & L
SPHL	1 1 1 1 1 0 0 1	H & L to stack pointer
LXI SP	0 0 1 1 0 0 0 1	Load immediate stack pointer
INX SP	0 0 1 1 0 0 1 1	Increment stack pointer
DCX SP	0 0 1 1 1 0 1 1	Decrement stack pointer
JUMP		
JMP	1 1 0 0 0 0 1 1	Jump unconditional
JC	1 1 0 1 1 0 1 0	Jump on carry
JNC	1 1 0 1 0 0 1 0	Jump on no carry
JZ	1 1 0 0 1 0 1 0	Jump on zero
JNZ	1 1 0 0 0 0 1 0	Jump on no zero
JP	1 1 1 1 0 0 1 0	Jump on positive
JM	1 1 1 1 1 0 1 0	Jump on minus
JPE	1 1 1 0 1 0 1 0	Jump on parity even
JPO	1 1 1 0 0 0 1 0	Jump on parity odd
PCHL	1 1 1 0 1 0 0 1	H & L to program counter
CALL		
CALL	1 1 0 0 1 1 0 1	Call unconditional
CC	1 1 0 1 1 1 0 0	Call on carry
CNC	1 1 0 1 0 1 0 0	Call on no carry
CZ		
CNZ	1 1 0 0 0 1 0 0	Call on no zero
CP	1 1 1 1 0 1 0 0	Call on positive
CM	1 1 1 1 1 1 0 0	Call on minus
CPE	1 1 1 0 1 1 0 0	Call on parity even
CPO	1 1 1 0 0 1 0 0	Call on parity odd
RETURN		
RET	1 1 0 0 1 0 0 1	Return
RC	1 1 0 1 1 0 0 0	Return on carry
RNC	1 1 0 1 0 0 0 0	Return on no carry
RZ	1 1 0 0 1 0 0 0	Return on zero
RNZ	1 1 0 0 0 0 0 0	Return on no zero
RP	1 1 1 1 0 0 0 0	Return on positive
RM	1 1 1 1 1 0 0 0	Return on minus
RPE	1 1 1 0 1 0 0 0	Return on parity even
RPO	1 1 1 0 0 0 0 0	Return on parity odd
RESTART		
RST	1 1 A A A 1 1 1	Restart
INPUT/OUTPUT		
IN	1 1 0 1 1 0 1 1	Input
OUT	1 1 0 1 0 0 1 1	Output
INCREMENT AND DECREMENT		
INR r	0 0 D D D 1 0 0	Increment register
DCR r	0 0 D D D 1 0 1	Decrement register
INR M	0 0 1 1 0 1 0 0	Increment memory
DCR M	0 0 1 1 0 1 0 1	Decrement memory
INX B	0 0 0 0 0 0 1 1	Increment B & C registers
INX D	0 0 0 1 0 0 1 1	Increment D & E registers
INX H	0 0 1 0 0 0 1 1	Increment H & L registers
DCX B	0 0 0 0 1 0 1 1	Decrement B & C
DCX D	0 0 0 1 1 0 1 1	Decrement D & E
DCX H	0 0 1 0 1 0 1 1	Decrement H & L
ADD		
ADD r	1 0 0 0 0 S S S	Add register to A
ADC r	1 0 0 0 1 S S S	Add register to A with carry
ADD M	1 0 C 0 0 1 1 0	Add memory to A
ADC M	1 0 0 0 1 1 1 0	Add memory to A with carry
ADI	1 1 0 0 0 1 1 0	Add immediate to A
ACI	1 1 0 0 1 1 1 0	Add immediate to A with carry
DAD B	0 0 0 0 1 0 0 1	Add B & C to H & L
DAD D	0 0 0 1 1 0 0 1	Add D & E to H & L
DAD H	0 0 1 0 1 0 0 1	Add H & L to H & L
DAD SP	0 0 1 1 1 0 0 1	Add stack pointer to H & L
SUBTRACT		
SUB r	1 0 0 1 0 S S S	Subtract register from A
SBB r	1 0 0 1 1 S S S	Subtract register from A with borrow
SUB M	1 0 0 1 0 1 1 0	Subtract memory from A
SBB M	1 0 0 1 1 1 1 0	Subtract memory from A with borrow
SUI	1 1 0 1 0 1 1 0	Subtract immediate from A
SBI	1 1 0 1 1 1 1 0	Subtract immediate from A with borrow

Table 6. Instruction Set Summary (Continued)

Mnemonic	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	Operations Description
LOGICAL									
ANA r	1	0	1	0	0	S	S	S	And register with A
XRA r	1	0	1	0	1	S	S	S	Exclusive OR register with A
ORA r	1	0	1	1	0	S	S	S	OR register with A
CMP r	1	0	1	1	1	S	S	S	Compare register with A
ANA M	1	0	1	0	0	1	1	0	And memory with A
XRA M	1	0	1	0	1	1	1	0	Exclusive OR memory with A
ORA M	1	0	1	1	0	1	1	0	OR memory with A
CMP M	1	0	1	1	1	1	1	0	Compare memory with A
ANI	1	1	1	0	0	1	1	0	And immediate with A
XRI	1	1	1	0	1	1	1	0	Exclusive OR immediate with A
ORI	1	1	1	1	0	1	1	0	OR immediate with A
CPI	1	1	1	1	1	1	1	0	Compare immediate with A
ROTATE									
RLC	0	0	0	0	0	1	1	1	Rotate A left
RRC	0	0	0	0	1	1	1	1	Rotate A right
RAL	0	0	0	1	0	1	1	1	Rotate A left through carry
RAR	0	0	0	1	1	1	1	1	Rotate A right through carry

Mnemonic	D <sub>7</sub>	D <sub>6</sub>	D <sub>5</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	Operations Description
SPECIALS									
CMA	0	0	1	0	1	1	1	1	Complement A
STC	0	0	1	1	0	1	1	1	Set carry
CMC	0	0	1	1	1	1	1	1	Complement carry
DAA	0	0	1	0	0	1	1	1	Decimal adjust A
CONTROL									
EI	1	1	1	1	1	0	1	1	Enable Interrupts
DI	1	1	1	1	0	0	1	1	Disable Interrupt
NOP	0	0	0	0	0	0	0	0	No-operation
HLT	0	1	1	1	0	1	1	0	Halt
NEW 8085A INSTRUCTIONS									
RIM	0	0	1	0	0	0	0	0	Read Interrupt Mask
SIM	0	0	1	1	0	0	0	0	Set Interrupt Mask

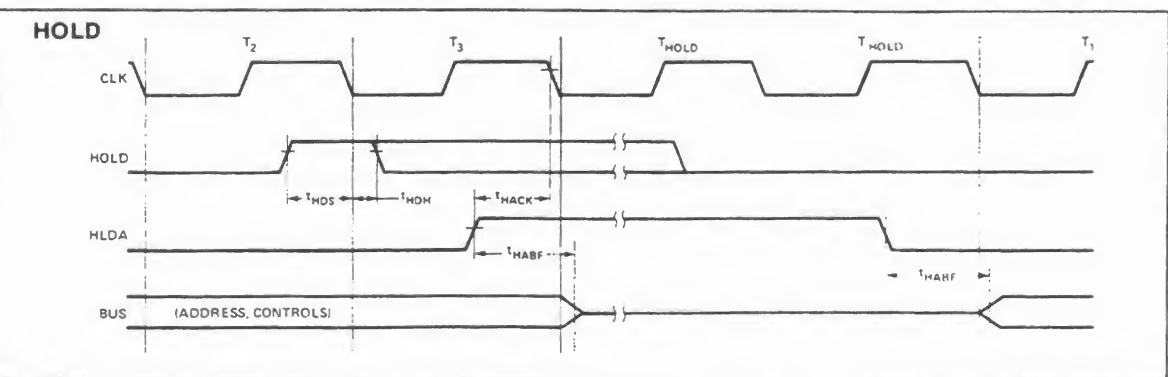
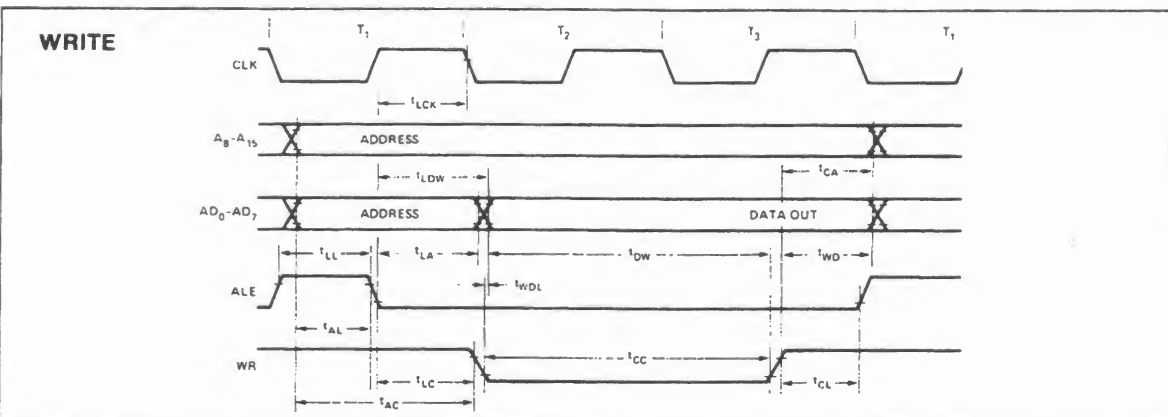
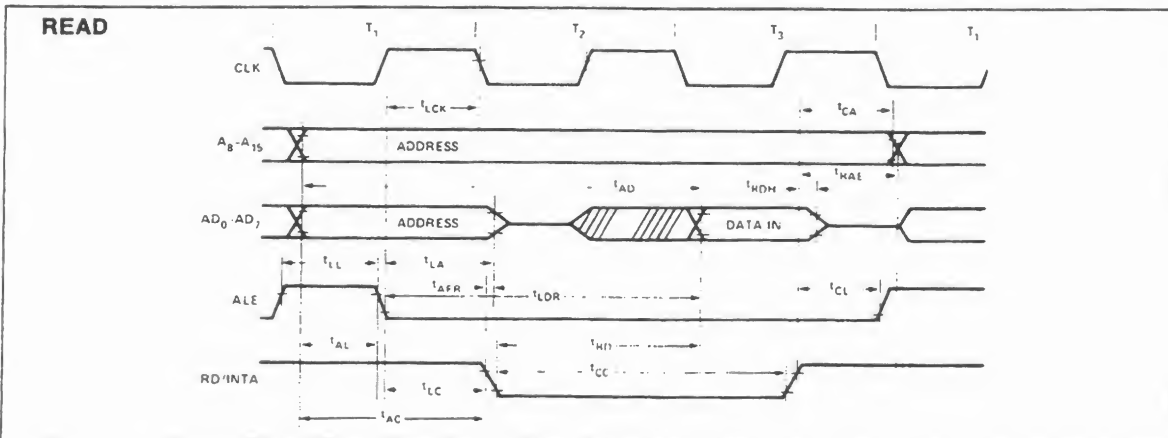
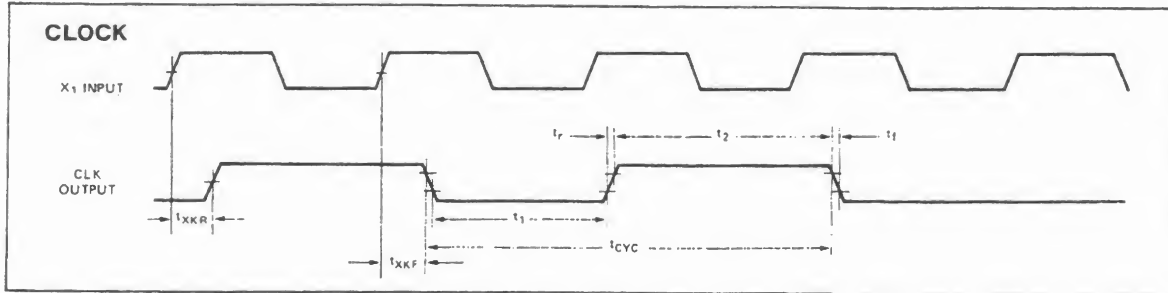
## NOTES:

1. DDS or SSS: B 000, C 001, D 010, E 011, H 100, L 101, Memory 110, A 111.
2. Two possible cycle times (6/12) indicate instruction cycles dependent on condition flags.

\*All mnemonics copyrighted © Intel Corporation 1976.



# WAVEFORMS







*Appendix C*

**PROGRAM CARTRIDGE  
SOURCE CODE LISTINGS**

## C-2 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

### File Digit 0

		PUTC	EQU	0046H
		SETRS232	EQU	0049
		SETLCD	EQU	004CH
7000	31 FF 8F		LXI SP,8FFFH	;Establish a stack
7003	21 34 70	BEGIN	LXI H,MESSAGE	;Point to message
7006	7E	GET	MOV A,M	;Get character
7007	FE 00		CPI 0	;Compare with 0
7009	CA 03 70		JZ BEGIN	;Start over if zero
700C	E5		PUSH H	;Save H-L
700D	F5		PUSH PSW	;Save PSW
700E	CD 49 00		CALL SETRS232	;Shift to RS232 mode
7011	CD 46 00		CALL PUTC	;Display Character
7014	CD 4C 00		CALL SETLCD	;Shift to LCD mode
7017	F1		POP PSW	;Restore PSW
7018	E1		POP H	;Restore H
7019	CD 46 00		CALL PUTC	;output character
701C	DB 90		IN 090H	;read data input port
701E	D3 A0		OUT 0A0H	;send byte to output port
7020	23		INX H	;point to next
7021	11 00 10		LXI D,1000H	;set up counter
7024	1B	DEC	DCX D	;decrement counter
7025	7B		MOV A,E	;copy count to a
7026	FE 00		CPI 0	;is high byte 0?
7028	C2 24 70		JNZ DEC	;decrement again
702B	7A		MOV A,D	;copy count to a
702C	FE 00		CPI 0	;is high byte 0?
702E	C2 24 70		JNZ DEC	;decrement again
7031	C3 06 70		JMP GET	;Do it all again
7034	38 30 38 35 20	MESSAGE	DB	"8085 Microprocessor",0DH
	4D 69 63 72 6F			
	70 72 6F 63 65			
	73 73 6F 72 0D			

## PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-3

---

7048	50 72 6F 67 72	DB	"Programming Course",ODH
	61 6D 6D 69 6E		
	67 20 43 6F 75		
	72 73 65 0D		
705B	42 79 20 4A 6F	DB	"By John D. Hubbard",ODH
	68 6E 20 44 2E		
	20 48 75 62 62		
	61 72 64 0D		
706E	43 6F 70 79 72	DB	"Copyright 1989",ODH
	69 67 68 74 20		
	31 39 38 39 0D		
707D	48 65 61 74 68	DB	"Heath Company",ODH
	20 43 6F 6D 70		
	61 6E 79 0D		
708B	42 65 6E 74 6F	DB	"Benton Harbor",ODH
	6E 20 48 61 72		
	62 6F 72 0D		
7099	4D 69 63 68 69	DB	"Michigan 49022",ODH,ODH,OOH
	67 61 6E 20 20		
	34 39 30 32 32		
	0D 0D 00		
70AB			END

## C-4 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

### File Digit 1

		GETC	EQU	0043H
		PUTC	EQU	0046H
7000	CD 43 00	START1	CALL	GETC;Get the key
7003	CD 46 00	CALL	PUTC	;Display it
7006	C3 00 70	JMP	START1	;Do it again
7009			DS	17H
		GETC	EQU	0043H
		PUTC	EQU	0046H
		STACK	EQU	6FFFH
7020	31 FF 6F	BEGIN1	LXI SP,STACK	;Establish stack
7023	CD 43 00		CALL GETC	;Get the key
7026	47		MOV B,A	;Save value
7027	1F		RAR	;Shift right one nibble
7028	1F		RAR	
7029	1F		RAR	
702A	1F		RAR	
702B	E6 0F		ANI 0FH	;Mask nibble
702D	CD 3E 70		CALL DOIT	;Display nibble
7030	78		MOV A,B	;Restore byte
7031	E6 0F		ANI 0FH	;Mask nibble
7033	CD 3E 70		CALL DOIT	;Display nibble
7036	3E 20		MVI A," "	;set up a space
7038	CD 46 00		CALL PUTC	;Display it
703B	C3 20 70		JMP BEGIN1	;Do it again
703E	FE 0A	DOIT	CPI 0AH	;Compare it to 0A
7040	FA 45 70		JM OK	;Jump if OK
7043	C6 07		ADI 07H	;Add 7 for letters
7045	C6 30	OK	ADI 30H	;Add 30 for all
7047	CD 46 00		CALL PUTC	;Display it
704A	C9		RET	;Return

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-5

704B		DS	0B5H
		GETC	EQU 0043H
		PUTC	EQU 0046H
		PSTRING	EQU 005BH
		STACK	EQU 6FFFH
7100	31 FF 6F	BEGIN	LXI SP,STACK ;Establish stack
7103	21 3D 71	LOOP	LXI H,MESSAGE;Point to the message
7106	CD 5B 00		CALL PSTRING ;Display it
7109	F5		PUSH PSW
710A	47		MOV B,A
710B	3E 30		MVI A,30H
710D	CE 00		ACI 0
710F	CD 46 00		CALL PUTC
7112	3E 20		MVI A," "
7114	CD 46 00		CALL PUTC
7117	CD 46 00		CALL PUTC
711A	78		MOV A,B
711B	0E 08		MVI C,8
711D	07	AFLAG	RLC
711E	5F		MOV E,A
711F	3E 30		MVI A,"0"
7121	CE 00		ACI 0
7123	CD 46 00		CALL PUTC
7126	7B		MOV A,E
7127	0D		DCR C
7128	C2 1E 71		JNZ AFLAG
712B	CD 43 00		CALL GETC
712E	FE 2D		CPI "-"
7130	CA 38 71		JZ RIGHT
7133	F1		POP PSW
7134	17		RAL
7135	C3 04 71		JMP LOOP
7138	F1	RIGHT	POP PSW
7139	1F		RAR

## C-6 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

```
713A  C3 03 71      JMP      LOOP
713D  0D 43 79 20 41 MESSAGE  DB      0DH,"Cy Accumulator",0DH,0
      63 63 75 6D 75
      6C 61 74 6F 72
      0D 00
714E                      END
```

## File Digit 2

7000	7E		MOV A,M	;Load a byte into A
7001	81		ADD C	;Add the two low bytes
7002	77		MOV M,A	;Store the result
7003	23		INX H	;Point to the next byte
7004	7E		MOV A,M	;Load the byte into A
7005	88		ADC B	;Add the next bytes
7006	77		MOV M,A	;Store the result
7007	23		INX H	;Point to the next byte
7008	7E		MOV A,M	;Load a byte into A
7009	8B		ADC E	;Add the next bytes
700A	77		MOV M,A	;Store the result
700B	23		INX H	;Point to the next byte
700C	7E		MOV A,M	;Load a byte into A
700D	8A		ADC D	;Add the high bytes
700E	77		MOV M,A	;Store the result
700F			DS	11H
7020	21 00 00		LXI H,0000	;Clear H-L
7023	06 00		MVI B,0	;Clear B
7025	09	LOOP1	DAD B	;Add B-C to H-L
7026	3D		DCR A	;Decrement A by 1
7027	C2 25 70		JNZ LOOP1	;Do again if A isn't 0
702A			DS	16H
7040	11 FF FF		LXI D,0FFFFH	;D=minus 1
7043	7D	LOOP	MOV A,L	;Get low byte into a
7044	91		SUB C	;Subtract low byte (no borrow)
7045	6F		MOV L,A	;Put difference back in l
7046	7C		MOV A,H	;Get high byte
7047	98		SBB B	;Subtract high byte (with borrow)
7048	67		MOV H,A	;Difference back to high
7049	13		INX D	;Count one subtraction (d=d+1)



## C-8 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

704A	D2 43 70		JNC LOOP	;Subtract again if no borrow
704D	09		DAD B	;Add divisor back to H-L if done
704E			DS	0B2H
		PUTC	EQU	0046H
		GETC	EQU	0043H
7100	CD 14 71	BEGIN	CALL	PROMPT;Output prompt, input value
7103	EB		XCHG	;Swap again
7104	22 12 71		SHLD	STORE2;Store the second value
7107	CD 44 71		CALL	SIGN;Input sign
710A	CD 5C 71		CALL	DISPLAY;Display result
710D	C3 00 71		JMP	BEGIN;Do it again
7110	00 00	STORE1	DB	00,00
7112	00 00	STORE2	DB	00,00
7114	3E 3E	PROMPT	MVI A, ""	;Set up an arrow
7116	CD 46 00		CALL PUTC	;Output character
7119	3E 20		MVI A, " "	;Set up a space
711B	CD 46 00		CALL PUTC	;Output character
711E	11 00 00		INPUT LXI	D,0;Set D-E to 0
7121	CD 43 00	NOTOK	CALL GETC	;Get a character
7124	CD 46 00		CALL PUTC	;Echo the character
7127	FE 2E		CPI 2EH	;Is this a sign?
7129	D8		RC	;Return if so
712A	D6 30		SUI 30H	;Subtract 30 to deASCII
712C	DA 21 71		JC NOTOK	;Try again if less than 0
712F	FE 0A		CPI 0AH	;Is it more than 9?
7131	D2 21 71		JNC NOTOK	;Try again if so
7134	21 00 00		LXI H,0	;Clear H for conversion
7137	19		DAD D	;H=D*1
7138	29		DAD H	;H=D*2
7139	29		DAD H	;H=D*4
713A	19		DAD D	;H=D*5
713B	29		DAD H	;H=D*10
713C	5F		MOV E,A	;Put new value in E
713D	16 00		MVI D,0	;Clear D

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-9

713F	19		DAD D	;Add new value
7140	EB		XCHG	;Put new value into D-E
7141	C3 21 71		jmp NOTOK	;Try again
7144	2A 12 71	SIGN	LHLD STORE2	;Get second value
7147	EB		XCHG	;Move it to D-E
7148	2A 10 71		LHLD STORE1	;Get first value
714B	FE 2D		CPI "-"	;Is it a minus?
714D	C2 57 71		JNZ ADD	;If not, add
				;If not, must be subtract
7150	7A		MOV A,D	;Move D to A
7151	2F		CMA	;Compliment D (in A)
7152	57		MOV D,A	;Restore it
7153	7A		MOV A,D	;Move E to A
7154	2F		CMA	;Compliment E (in A)
7155	5F		MOV E,A	;Restore it
7156	13		INX D	;Make 2's compliment
7157	19		ADD DAD	D;Add them
7158	22 10 71		SHLD STORE1	;Store result
715B	C9		RET	
715C	3E 0D	DISPLAY	MVI A,0DH	;Carriage return first
715E	E5		PUSH H	
715F	CD 46 00		CALL PUTC	;Output character
7162	7C		MOV A,H	;Sample high byte
7163	FE 80		CPI 80H	;Test high bit
7165	3E 00		MVI A,0	;Clear A
7167	FA 73 71		JM OK	;If minus, its ok
716A	7C		MOV A,H	;Move H to A
716B	2F		CMA	;Compliment H (in A)
716C	67		MOV H,A	;Restore it
716D	7D		MOV A,L	;Move L to A
716E	2F		CMA	;Compliment L (in A)
716F	6F		MOV L,A	;Restore it
7170	23		INX H	;Make 2's compliment
7171	3E 02		MVI A,2	;This will make it minus
7173	C6 2B		OK ADI	2Bh;+ or -

## C-10 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

7175	CD 46 00		CALL PUTC	;Output character
7178	01 10 27		LXI B,10000	;Load B with 10000
717B	CD 98 71		CALL SUBTR	;Get ten thousands
717E	01 E8 03		LXI B,1000	;Load B with 1000
7181	CD 98 71		CALL SUBTR	;Get thousands
7184	01 64 00		LXI B,100	;Load B with 100
7187	CD 98 71		CALL SUBTR	;Get hundreds
718A	01 0A 00		LXI B,10	;Load B with 10
718D	CD 98 71		CALL SUBTR	;Get tens
7190	7D		MOV A,L	;Units
7191	C6 30		ADI 30H	;Make ASCII
7193	CD 46 00		CALL PUTC	;Display it
7196	E1		POP H	
7197	C9		RET	
7198	16 FF	SUBTR	MVI D,0FFH	;D=minus 1
719A	7D	LOOPY	MOV A,L	;Get low byte
719B	91		SUB C	;Subtract low byte
719C	6F		MOV L,A	;Put difference back
719D	7C		MOV A,H	;Get high byte
719E	98		SBB B	;Subtract high byte
719F	67		MOV H,A	;Difference to high
71A0	14		INR D	;Count one subtraction
71A1	D2 9A 71		JNC LOOPY	;Subtract again if no borrow
71A4	09		DAD B	;Add divisor to unknown if done
71A5	7A		MOV A,D	;Put count in A
71A6	C6 30		ADI 30H	;Make it ASCII
71A8	E5		PUSH H	;Save the pointer
71A9	CD 46 00		CALL PUTC	;Display the value
71AC	E1		POP H	;Restore the pointer
71AD	C9		RET	
71AE			END	

**File Digit 3**

7000	3E 47	MVI A,47H	;put value (47) in A
7002	E6 F0	ANI 0F0H	;mask high nibble
7004	C6 05	ADI 5	;Add 5 to Reg A
7006	E6 0F	ANI 0FH	;mask low nibble
7008	3E 47	MVI A,47H	;put value (47) in A
700A	06 F0	MVI B,0F0H	;Mask of B's high nibble
700C	A0	ANA B	;Apply the mask
700D	C6 05	ADI 5	;Add 5 to Reg A
700F	06 0F	MVI B,0FH	;Mask of B's low nibble
7011	A0	ANA B	;Apply the mask
7012		DS 0eH	
7020	3E 07	MVI A,07H	;Put 7 into Accumulator
7022	F6 40	ORI 40H	;Apply 4 to high nibble
7024	F6 0F	ORI 0FH	;Conceal low nibble
7026	E6 F3	ANI 0F3H	;Convert low nibble
7028	F6 FF	ORI 0FFH	;Conceal whole byte
702A	06 65	MVI B,65H	;Set up value
702C	A0	ANA B	;Convert whole byte
702D	06 0A	MVI B,0AH	;Establish mask
702F	B0	ORA B	;Fill in bits
7030		ds 10H	
7040	3E 47	MVI A,47H	;Put 47 into A
7042	EE 0F	XRI 0FH	;XOR with 0F
7044	EE 0F	XRI 0FH	;XOR with 0F
7046	3E 0F	MVI A,0FH	;Put 0F into A
7048	EE FF	XRI 0FFH	;XOR with FF
704A	EE A5	XRI 0A5H	;XOR with A5
704C		DS 14H	

## C-12 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

7060	3E 47	MVI A,47H	;Put 47 in Accumulator
7062	C6 33	ADI 33H	;Add 33
7064	27	DAA	;Decimal adjust
7065	C6 98	ADI 98H	;Add 98
7067	27	DAA	;Decimal adjust
7068	D6 70	SUI 70H	;Subtract 70
706A	87	ADD A	;Add Accumulator
706B	27	DAA	;Decimal adjust
706C	C6 40	ADI 40H	;Add 40
706E	27	DAA	;Decimal adjust
706F	C6 44	ADI 44H	;Add 44
7071	27	DAA	;Decimal adjust
7072		DS 0eH	
7080	3E 0F	MVI A,0FH	;put 0F in accumulator
7082	2F	CMA	;Compliment A
7083	3E 33	MVI A,33H	;Put 33 in Accumulator
7085	2F	CMA	;Compliment A
7086	C6 47	ADI 47H	;Add 47
7088	3C	INR A	;Add 1
7089		DS 07H	
7090	3E 61	MVI A,61H	;put "a" in the accumulator
7092	E6 DF	ANI 0DFH	;mask to upper case
7094	F6 20	ORI 20H	;mask back to lower case
7096	EE 20	XRI 20H	;Convert to other case
7098	EE 20	XRI 20H	;Convert to other case
709A		DS 06H	
70A0	3E 55	MVI A,55H	;55 alternates ones and zeros
70A2	0F	RRC	;Everybody move right.
70A3	0F	RRC	;Everybody move right.
70A4	0F	RRC	;Everybody move right.
70A5	0F	RRC	;Everybody move right.

## PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-13

---

70A6	2F	CMA	;Reverse the pattern.
70A7	07	RLC	;Everybody move left.
70A8	07	RLC	;Everybody move left.
70A9	07	RLC	;Everybody move left.
70AA	07	RLC	;Everybody move left.
70AB		END	

## C-14 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

### File Digit 4

7000	0E 10	MVI C,10H	;Set counter reg C to 10H
7002	21 00 80	LXI H,8000H	;Point the H-L pair to 8000H.
7005	36 00	MVI M,00	;Put 00 into memory M.
7007	23	INX H	;Increment H-L.
7008	0D	DCR C	;Decrement the counter.
7009	C2 05 70	JNZ 7005H	;Repeat if count not 0
700C	CD 40 00	CALL 0040H	;Return to monitor
700F		DS 01H	
7010	21 00 80	LXI H,8000H	;Point to 8000H
7013	36 5A	MVI M,5AH	;Put Z at 8000H
7015	66	MOV H,M	;Copy M to H
7016	6C	MOV L,H	;Copy H to L
7017	5D	MOV E,L	;Copy L to E
7018	53	MOV D,E	;Copy E to D
7019	4A	MOV C,D	;Copy D to C
701A	41	MOV B,C	;Copy C to B
701B	78	MOV A,B	;Copy B to A
701C		DS 04H	
7020	06 80	MVI B,80H	;Put 80H in B
7022	0E 38	MVI C,38H	;Put 38H in C
7024	79	MOV A,C	;Copy C to A
7025	17	RAL	;Rotate left
7026	4F	MOV C,A	;Put back in C
7027	78	MOV A,B	;Copy B to A
7028	17	RAL	;rotate high to CY
7029	79	MOV A,C	;Copy C to A
702A	1F	RAR	;rotate carry right
702B		DS 05H	

## PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-15

---

7030	06 80	MVI B,80H	;Put 80H in B
7032	0E 38	MVI C,38H	;Put 38H in C
7034	79	MOV A,C	;Copy C to A
7035	E6 7F	ANI 7FH	;Mask high bit to 0
7037	4F	MOV C,A	;Put back in C
7038	78	MOV A,B	;Copy B to A
7039	E6 80	ANI 80H	;Mask low 7 bits to 0's
703B	B1	ORA C	;OR C with A
703C		END	



## File Digit 5

7000	2A 01 70	LHLD 7001H	;Load pointer from 7001
7003	3A 00 70	LDA 7000H	;Load A from 7000
7006	32 0C 70	STA 700CH	;Store A at 700C
7009	22 0D 70	SHLD 700DH	;Store pointer at 700D
700C		DS 014H	
7020	01 40 70	LXI B,7040H	;Point B at 7040
7023	11 41 70	LXI D,7041H	;Point D at 7041
7026	3E 48	MVI A,48H	;Set A to 48
7028	02	STAX B	;Store A at 7040
7029	3E 5A	MVI A,5AH	;Set A to 5A
702B	12	STAX D	;Store A at 7041
702C	AF	XRA A	;Clear A
702D	03	INX B	;Point B at 7041
702E	1B	DCX D	;Point D at 7040
702F	0A	LDAX B	;Retrieve value at 7041
7030	1A	LDAX D	;Retrieve value at 7040
7031		DS 01FH	
7050	21 34 12	LXI H,1234H	;Place 1234 into H-L
7053	22 00 80	SHLD 8000H	;Store 1234 at 8000
7056	21 00 80	LXI H,8000H	;Point to 8000 with H-L
7059	F9	SPHL	;Transfer pointer to SP
705A	21 78 56	LXI H,5678H	;Place 5678 into H-L
705D	E3	XTHL	;Exchange H-L with 8000
705E	2A 00 80	LHLD 8000H	;Load H-L from 8000
7061	11 50 70	LXI D,7050H	;Place 7050 into D-E
7064	EB	XCHG	;Exchange H-L and D-E
7065	E9	PCHL	;Indirect jump to 7050

## File Digit 6

7000	3E 4A		MVI A,"J"	;Put the letter "J" into A
7002	CD 46 00		CALL 0046H	;Display the letter
7005	3E 44		MVI A,"D"	;Put the letter "D" into A
7007	CD 46 00		CALL 0046H	;Display the letter
700A	3E 48		MVI A,"H"	;Put the letter "H" into A
700C	CD 46 00		CALL 0046H	;Display the letter
700F	3E 20		MVI A,20H	;Put a space into A
7011	CD 46 00		CALL 0046H	;Display the space
7014	C3 00 70		JMP 7000H	;Jump to 7000
7017			DS 09H	
7020	21 29 70		LXI H,7029H	;Point to the data
7023	CD 5B 00		CALL 005BH	;Display the data
7026	C3 3D 70		JMP AROUND	;Jump around the data
7029	54 68 69 73		DB "This"	;Data
702D	20 69 73 20		DB " is "	;Data
7031	61 20 6D 65		DB "a me"	;Data
7035	73 73 61 67		DB "ssag"	;Data
7039	65 2E 20 00		DB "e. ",0	;Data
703D	00	AROUND	NOP	;Do nothing
703E	C3 3E 70	SELF	JMP SELF	;Hold here
7041			DS 0FH	
7050	AF	START	XRA A	;Set Z & P, Clear S & CY
7051	CA 60 70		JZ PT1	;Jump if Zero
7054	F2 63 70	PT5	JP PT2	;Jump if Positive
7057	3C	PT6	INR A	;Clear Z & P
7058	C2 66 70		JNZ PT3	;Jump if Not Zero
705B	D6 02	PT7	SUI 2	;Set S & CY
705D	DA 69 70	PT8	JC PT4	;Jump if Carry
7060	EA 54 70	PT1	JPE PT5	;Jump if even Parity
7063	D2 57 70	PT2	JNC PT6	;Jump if no carry

## C-18 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

7066	E2 5B 70	PT3	JPO PT7	;Jump if odd Parity
7069	FA 50 70	PT4	JM START	;Jump if minus
706C			DS 04H	
7070	21 00 40	BEGIN	LXI H,4000H	;Point to First address
7073	7E	LOOP	MOV A,M	;Get value from memory
7074	CD 46 00		CALL 0046H	;Display it
7077	BE		CMP M	;Compare A to memory value
7078	C2 88 70		JNZ FOUND	;Jump if FOUND different
707B	11 01 90		LXI D,9001H	;Set D to 6FFF compliment
707E	EB		XCHG	;Trade D-E and H-L
707F	19		DAD D	;Add D-E to H-L
7080	DA 8B 70		JC FINISHED	;Jump if FINISHED
7083	EB		XCHG	;Trade back if not
7084	23		INX H	;Increment pointer
7085	C3 73 70		JMP LOOP	;Repeat
7088	CD 58 00	FOUND	CALL 0058H	;Display address
708B	C3 8B 70	FINISHED	JMP FINISHED	;Loop to self forever
708E			END	

## File Digit 7

	BS	EQU 8	;BS means Back Space
	GETC	EQU 0043H	;Get Character
	PUTC	EQU 0046H	;Put Character
7000	CD 43 00	BEGIN	CALL GETC ;Get a character
7003	FE 2B		CPI "+" ;Is it "+"?
7005	CC 0E 70		CZ UPDN ;Call up/down if so
7008	CD 46 00		CALL PUTC ;Display the character
700B	C3 00 70		JMP BEGIN ;Do it all again
700E	21 FF 7F	UPDN	LXI H,7FFFH ;Point to safe memory
7011	36 50		MVI M,"P" ;Put a "P" into memory
7013	7E	AGAIN	MOV A,M ;Set up the letter
7014	CD 46 00		CALL PUTC ;Display it
7017	3E 08		MVI A,BS ;Set up a backspace
7019	CD 46 00		CALL PUTC ;Display the backspace
701C	CD 43 00		CALL GETC ;Get another key
701F	FE 3F		CPI "?" ;Is it Help?
7021	CA 2F 70		JZ DONE ;If so, we're done
7024	FE 2B		CPI "+" ;Is it "+"
7026	CA 2B 70		JZ PLUS1 ;Add one if so
7029	35		DCR M ;Must be minus
702A	35		DCR M ;Do this twice to save space
702B	34	PLUS1	INR M ;Add one to character
702C	C3 13 70		JMP AGAIN ;Do it again
702F	7E	DONE	MOV A,M ;Set up the character
7030	C9		RET ;Return
7031			END

## C-20 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

### File Digit 8

7000	31 FF 6F	BEGIN	LXI SP,6FFFH ;Put stack at 6FFF
7003	21 22 70		LXI H,TARGET ;Put a value into H-L
7006	E5		PUSH H ;Store it on the stack
7007	E5		PUSH H ;Store it again
7008	CD 11 70		CALL RETURN ;Call a subroutine
700B	D1		POP D ;Take from stack put in D-E
700C	F1		POP PSW ;Take from stack put in PSW
700D	F5		PUSH PSW ;Put PSW on stack
700E	C1		POP B ;Take from stack put in B-C
700F	3B		DCX SP ;Decrement the stack pointer
7010	3B		DCX SP ;Decrement the stack pointer
7011	C9	RETURN	RET ;A one step subroutine
7012	C3 00 70		JMP BEGIN ;Go to the beginning
7015			DS 0DH
7022	CD 40 70	TARGET	CALL SHOWPC ;Display the calling address
7025	C3 00 70		JMP BEGIN ;Go to the beginning
7028			DS 018H
7040	E1	SHOWPC	POP H ;Get the return address
			;from the stack into H.
7041	3B		DCX SP ;Correct the SP so return
7042	3B		DCX SP ; address is back on top.
7043	2B		DCX H ;Adjust for three bytes
7044	2B		DCX H ; in the
7045	2B		DCX H ; CALL instruction
7046	CD 58 00		CALL 0058H ;Display H-L
7049	CD 43 00		CALL 0043H ;Pause here for a key
704C	C9		RET ;Return to source
704D			DS 023H

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-21

7070	CD 40 70		CALL SHOWPC	;Display the calling address
7073	C3 00 70		JMP BEGIN	;Go to the beginning
7076			DS 08AH	
		BS	EQU 8	;BS means Back Space
		GETC	EQU 0043H	;Get Character
		PUTC	EQU 0046H	;Put Character
		STACK	EQU 7FFEh	;We'll put the stack here
7100	31 FE 7F	START	LXI SP,STACK	;Establish a stack
7103	3E 00		MVI A,0	;Set up a delimiter
7105	F5		PUSH PSW	;Put it on the stack
7106	3E 0D		MVI A,0DH	;Set up a carriage Return
7108	F5		PUSH PSW	;Put that on the stack
7109	CD 43 00	INPUT	CALL GETC	;Get a character
710C	FE 2B		CPI "+"	;Is it a "+"?
710E	CC 1D 71		CZ UPDN	;If so, call ud/down
7111	CD 46 00		CALL PUTC	;Display it
7114	FE 3F		CPI "?"	;Is it RPO?
7116	CA 40 71		JZ PLAYBAK	;Play it back
7119	F5		PUSH PSW	;Save the character
711A	C3 09 71		JMP INPUT	;Get the next Character
711D	21 FF 7F	UPDN	LXI H,7FFFH	;point to memory
7120	36 50		MVI M,"P"	;Put a "P" into memory
7122	7E	AGAIN	MOV A,M	;Set up the letter
7123	CD 46 00		CALL PUTC	;Display it
7126	3E 08		MVI A,BS	;Set up a backspace
7128	CD 46 00		CALL PUTC	;Display the backspace
712B	CD 43 00		CALL GETC	;Get another key
712E	FE 3F		CPI "?"	;Is it Help?
7130	CA 3E 71		JZ DONE	;If so, we're done
7133	FE 2B		CPI "+"	;Is it "+"
7135	CA 3A 71		JZ PLUS1	;Add one if so
7138	35		DCR M	;Must be minus

## C-22 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

7139	35		DCR M	;Do this twice to save space
713A	34	PLUS1	INR M	;Add one to character
713B	C3 22 71		JMP AGAIN	;Do it again
713E	7E	DONE	MOV A,M	;Set up the character
713F	C9		RET	;Return
7140	F1	PLAYBAK	POP PSW	;Get a character off stack
7141	E6 FF		ANI OFFH	;Is it the delimiter?
7143	CA 00 71		JZ START	;If so, start all over
7146	CD 46 00		CALL PUTC	;If not, display it
7149	C3 40 71		JMP PLAYBAK	;Do it again
714C			END	

## File Digit 9

7000	21 0E 70		LXI H,MESSG	;Point to the message
7003	CD 5B 00		CALL 005BH	;Display the message
7006	CD 67 00		CALL 0067H	;Wait a moment
7009	C9		RET	;Go to address after RST
700A	CF	LOOP	RST 1	
700B	C3 0A 70		JMP LOOP	;Do it again
700E	54 68 69 73	MESSG	DB "This"	
7012	20 69 73 20		DB " is "	
7016	74 68 65 20		DB "the "	
701A	70 72 6F 67		DB "prog"	
701E	72 61 6D 2E		DB "ram."	
7022	00		DB 0	
7023			END	



## File Digit A

7000	31 FF 6F	PUTC	EQU 46H	
7003	DB 00	BEGIN	LXI SP,6FFFH	;Establish a stack
7005	FE FF		IN 00	;Get Keyboard column 1
7007	CA 03 70		CPI 0FFH	;Is a key pressed
700A	47		JZ BEGIN	;IF NOT FORGET IT
700B	DB 00		MOV B,A	;Save this value
700D	FE FF	L0	IN 00	;Get coll again
700F	C2 0B 70		CPI 0FFH	;Is key released?
7012	78		JNZ L0	;Loop until done
7013	CD 19 70		MOV A,B	;Restore the value
7016	C3 03 70		CALL FINDIT	;Find and display it.
7019	21 2D 70		JMP BEGIN	;Try again if not
701C	A7	FINDIT	LXI H, TABLE	;Point to the table
701D	07		ANA A	;Clear the Carry
701E	F5	LOOP	RLC	;Rotate into the carry
701F	D2 27 70		PUSH PSW	;Save the value
7022	23		JNC FOUND	;Jump if found
7023	F1		INX H	;Point to the next item
7024	C3 1D 70		POP PSW	;Restore the value
7027	F1		JMP LOOP	;Try again
7028	7E	FOUND	POP PSW	;Clean up the stack
7029	CD 46 00		MOV A,M	;Read the table
702C	C9		CALL PUTC	;Display it
702D	00	TABLE	RET	;Return
702E	33		DB 00H	;High bit Col 1 is clear
702F	36		DB 33H	;3 key
7030	39		DB 36H	;6 key
7031	43		DB 39H	;9 key
7032	46		DB 43H	;C key
7033	3F		DB 46H	;F key
			DB 3FH	;? key
			DS 0CCH	

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-25

		PUTC	EQU	46H
7100	31 FF 6F		LXI SP,6FFFH	;Establish a stack
7103	DB 00	BEGIN1	IN 00	;Get Keyboard column 1
7105	FE FF		CPI OFFH	;Is a key pressed
7107	CA 18 71		JZ COLM2	;If Not try column 2
710A	21 5A 71		LXI H,TABLE1	;Point to the table
710D	47		MOV B,A	;Save this value
710E	DB 00	L1	IN 00H	;Get coll again
7110	FE FF		CPI OFFH	;Is the key up?
7112	CA 42 71		JZ UPKEY	;If so, we're done
7115	C3 0E 71		JMP L1	;Loop until key up
7118	DB 10	COLM2	IN 10H	;Get Keyboard column 2
711A	FE FF		CPI OFFH	;Is a key pressed
711C	CA 2D 71		JZ COLM3	;If so try column 3
711F	21 61 71		LXI H,TABLE2	;Point to the table
7122	47		MOV B,A	;Save this value
7123	DB 10	L2	IN 10H	;Get col2 again
7125	FE FF		CPI OFFH	;Is the key up?
7127	CA 42 71		JZ UPKEY	;If so, we're done
712A	C3 23 71		JMP L2	;Loop until key up
712D	DB 20	COLM3	IN 20H	;Get Keyboard column 3
712F	FE FF		CPI OFFH	;Is a key pressed
7131	CA 03 71		JZ BEGIN1	;If NOT, try again
7134	21 68 71		LXI H,TABLE3	;Point to the table
7137	47		MOV B,A	;Save this value
7138	DB 20	L3	IN 20H	;Get col3 again
713A	FE FF		CPI OFFH	;Is the key up?
713C	CA 42 71		JZ UPKEY	;If so, we're done
713F	C3 38 71		JMP L3	;Loop until key up
7142	78	UPKEY	MOV A,B	;Restore the value
7143	CD 49 71		CALL FINDIT1	;Find and display it
7146	C3 03 71		JMP BEGIN1	;Try again if not
7149	A7	FINDIT1	ANA A	;Clear the Carry
714A	07	LOOP1	RLC	;Rotate into the carry
714B	F5		PUSH PSW	;Save the value
714C	D2 54 71		JNC FOUND1	;Jump if found
714F	23		INX H	;Point to the next item

## C-26 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

7150	F1		POP PSW	;Restore the value
7151	C3 4A 71		JMP LOOP1	;Try again
7154	F1	FOUND1	POP PSW	;Clean up the stack
7155	7E		MOV A,M	;Read the table
7156	CD 46 00		CALL PUTC	;Display it
7159	C9		RET	;Return
715A	00	TABLE1	DB 00H	;High bit Col 1 is clear
715B	33		DB 33H	;3 key
715C	36		DB 36H	;6 key
715D	39		DB 39H	;9 key
715E	43		DB 43H	;C key
715F	46		DB 46H	;F key
7160	3F		DB 3FH	;? key
7161	00	TABLE2	DB 00H	;High bit Col 1 is clear
7162	32		DB 32H	;2 key
7163	35		DB 35H	;5 key
7164	38		DB 38H	;8 key
7165	42		DB 42H	;B key
7166	45		DB 45H	;E key
7167	2B		DB 2BH	;+ key
7168	30	TABLE3	DB 30H	;0 KEY
7169	31		DB 31H	;1 key
716A	34		DB 34H	;4 key
716B	37		DB 37H	;7 key
716C	41		DB 41H	;A key
716D	44		DB 44H	;D key
716E	2D		DB 2DH	; - key
716F			END	

## File Digit B

		PUTC	EQU 0046H	;Routine to display character
		INTR	EQU 6816H	;Vector address for RST ____
7000	31 FF 6F	BEGIN	LXI SP,6FFFH	;Establish a stack
7003	3E FF		MVI A,OFFH	;Set all bits high
7005	D3 A0		OUT 0A0H	;Turn on the LEDs
7007	21 1F 70		LXI H,INTRUPT	;Point to interrupt routine
700A	22 16 68		SHLD INTR	;Change the vector
700D	21 38 70	AGAIN	LXI H,MSG1	;Point to the message
7010	7E	LOOP	MOV A,M	;Get the character
7011	A7		ANA A	;Is it zero?
7012	CA 0D 70		JZ AGAIN	;If so, start over
7015	23		INX H	;Point to next character
7016	CD 46 00		CALL PUTC	;Display it
7019	CD 29 70		CALL PAUSE	;Wait a while
701C	C3 10 70		JMP LOOP	;Go back for next character
701F	D3 A0	INTRUPT	OUT 0A0H	;Display ASCII on LEDs
7021	CD 29 70		CALL PAUSE	;Wait a while
7024	FB		EI	;Reenable the interrupts
7025	C9		RET	;Return for service routine
7026	C3 0D 70		JMP AGAIN	;For Step 9
7029	E5	PAUSE	PUSH H	;Save H
702A	D5		PUSH D	;Save D
702B	11 01 00		LXI D,1	;Set up increment
702E	21 00 00		LXI H,0	;Clear counter
7031	19	DLAY	DAD D	;Increment H
7032	D2 31 70		JNC DLAY	;If carry, we're done
7035	D1		POP D	;Restore D
7036	E1		POP H	;Restore H
7037	C9		RET	;Pause routine done
7038	53 61 6D 65 20	MSG1	DB "Same "	;This is the message
	6F 6C 64 20 6D		DB "old m"	
	65 73 73 61 67		DB "essag"	
	65 0D 00		DB "e",0DH,0	

## C-28 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

### File Digit C

		PUTC	EQU 0046H	;Display character routine
		RST55	EQU 6813H	;RST 5.5 vector
		COUNT	EQU 10000	;Count
7000	31 FF 6F	BEGIN	LXI SP,6FFFH	;Establish Stack
7003	20		RIM	;Read the interrupt mask
7004	E6 06		ANI 06H	;Add RST 5.5 to mask
7006	F6 08		ORI 08H	;Add mask set enable (mse)
7008	30		SIM	;Establish new mask
7009	21 21 70		LXI H,INTRUPT	;Point to new routine
700C	22 13 68		SHLD RST55	;Replace old vector
700F	21 56 70	AGAIN	LXI H,MSG1	;Point to message
7012	7E	LOOP	MOV A,M	;Get the character
7013	A7		ANA A	;Set the flags
7014	CA 0F 70		JZ AGAIN	;If message is done, repeat
7017	CD 46 00		CALL PUTC	;Display character
701A	23		INX H	;Point to next character
701B	CD 40 70		CALL DELAY	;Pause a while
701E	C3 12 70		JMP LOOP	;G&back for next character
7021	20	INTRUPT	RIM	;Get the mask
7022	E6 06		ANI 06H	;Save RST 7.5 and RST 6.5
7024	F6 09		ORI 09H	;Add mse & mask for RST 5.5
7026	30		SIM	;Mask RST 5.5
7027	E5		PUSH H	;Save H-L
7028	21 68 70		LXI H,MSG	;Point to RST 5.5 message
702B	7E	LP	MOV A,M	;Get the character
702C	A7		ANA A	;Set the flags
702D	CA 3A 70		JZ DONE	;Jump if message is done
7030	CD 46 00		CALL PUTC	;Display it
7033	23		INX H	;Point to next character
7034	CD 40 70		CALL DELAY	;Pause a while
7037	C3 2B 70		JMP LP	;Repeat for next character
703A	3E 08	DONE	MVI A,08H	;We'll reenale interrupts

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-29

```

703C 30          SIM          ;Clear RST interrupt mask
703D E1          POP H        ;Restore H-L
703E FB          EI          ;Reenable interrupts
703F C9          RET          ;Interrupt routine complete

7040 D5          DELAY        PUSH D      ;Save D
7041 F5          PUSH PSW     ;Save PSW
7042 C5          PUSH B       ;Save B
7043 06 01        MVI B,1     ;Load count multiplier
7045 11 10 27     TIMER      LXI D,COUNT ;Set counter
7048 1B          LP3         DCX D        ;Decrement count
7049 7A          MOV A,D      ;load high byte to A
704A B3          ORA E        ;Are both zero?
704B C2 48 70     JNZ LP3     ;If so, timer is done
704E 05          DCR B        ;Decrement multiplier
704F C2 45 70     JNZ TIMER   ;Repeat till Multiplier=0
7052 C1          POP B        ;Restore B
7053 F1          POP PSW     ;Restore PSW
7054 D1          POP D        ;Restore D
7055 C9          RET          ;Delay done

7056 53 61 6D 65 20 MSG1     DB "Same old message",0DH,0
      6F 6C 64 20 6D
      65 73 73 61 67
      65 0D 00
7068
7068 7B 69 6E 74 20 MSG     DB "{int 5.5}",00
      35 2E 35 7D 00
7072          END

```

## File Digit D-E-F (WISE)

	CR	EQU	00DH
	LF	EQU	00AH
	DEL	EQU	008H
	BSPACE	EQU	008H
	GETC	EQU	0043H
	PUTC	EQU	0046H
	SETRS232	EQU	0049H
	SETLCD	EQU	004CH
	GETHBYTE	EQU	004FH
	GETHWORD	EQU	0052H
	PUTHBYTE	EQU	0055H
	PUTHWORD	EQU	0058H
	PUTSTRING	EQU	005BH
	COPYUP	EQU	005EH
	COPYDOWN	EQU	0061H
	COPY	EQU	0064H
	PAUSE	EQU	0067H
	PUTDSWRD	EQU	006AH
	PUTUSWRD	EQU	006DH
	MONIT	EQU	0040H
	STACK	EQU	6FFFH
7000	31 FF 6F	BEGIN	LXI SP, STACK
7003	CD 6E 73	CALL	SHOTEL
7006	57 20 2A 20 49	DB	"W * I * S * E", CR, LF, 00
	20 2A 20 53 20		
	2A 20 45 0D 0A		
	00		
7016			
7016	06 0A	MVI	B, 00AH
7018	21 9A 73	LXI	H, PCSTOR
701B	36 00	ZLOOP	MVI M, 000H
701D	23	INX	H
701E	05	DCR	B

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-31

701F	C2 1B 70		JNZ	ZLOOP
7022	21 00 80		LXI	H, 8000H
7025	22 A4 73		SHLD	SPSTOR
7028	C3 6C 72		JMP	SPREAD
702B	21 A6 73	NEXT1	LXI	H, BUFFR
702E	31 FF 6F		LXI	SP, STACK
7031	CD 43 00	INLOOP	CALL	GETC
7034	FE 08		CPI	DEL
7036	CA 46 70		JZ	ERASE
7039	CD 46 00		CALL	PUTC
703C	77		MOV	M, A
703D	23		INX	H
703E	FE 0D		CPI	CR
7040	CA 55 70		JZ	DONEIN
7043	C3 31 70		JMP	INLOOP
7046	3E A6	ERASE	MVI	A, BUFFR ;LOW BYTE ONLY
7048	BD		CMP	L
7049	CA 31 70		JZ	INLOOP
704C	2B		DCX	H
704D	3E 08		MVI	A, BSPACE
704F	CD 46 00		CALL	PUTC
7052	C3 31 70		JMP	INLOOP
7055	21 B6 73	DONEIN	LXI	H, TABLE
7058	06 00		MVI	B, 000H
705A	11 A5 73	FINDIT	LXI	D, BUFFR-1
705D	13	KEEPON	INX	D
705E	1A		LDAX	D
705F	BE		CMP	M
7060	7E		MOV	A, M
7061	23		INX	H
7062	CA 5D 70		JZ	KEEPON
7065	FE 20		CPI	020H
7067	DA 77 70		JC	FOUND



## C-32 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

706A	7E	PASSBY	MOV	A, M
706B	23		INX	H
706C	FE 20		CPI	020H
706E	D2 6A 70		JNC	PASSBY
7071	04		INR	B
7072	C2 5A 70		JNZ	FINDIT
7075	3E 01		MVI	A, 001H
7077	F5	FOUND	PUSH	PSW
7078	78		MOV	A, B
7079	32 45 72		STA	OPCODE
707C	32 5E 73		STA	CONDTN
707F	21 00 00		LXI	H, 00000H
7082	22 46 72		SHLD	ARGMNT
7085	22 5F 73		SHLD	ADDRES
7088	F1		POP	PSW
7089	3D		DCR	A
708A	C2 A7 70		JNZ	TYPE2
708D	CD 6E 73		CALL	SHOTEL
7090	0D 20 49 6E 76		DB	CR, " INVALID MNEMONIC ", 0
	61 6C 69 64 20			
	6D 6E 65 6D 6F			
	6E 69 63 20 00			
70A4	C3 25 73		JMP	NEXT2
70A7	3D	TYPE2	DCR	A
70A8	C2 C6 70		JNZ	TYPE3
70AB	0E 01		MVI	C, 1
70AD	1A		LDAX	D
70AE	FE 0D		CPI	CR
70B0	CA 2C 72		JZ	DOINSTR
70B3	CD 6E 73	BADOP	CALL	SHOTEL
70B6	0D 62 61 64 20		DB	CR, "BAD OPERAND", 0
	4F 70 65 72 61			
	6E 64 00			

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-33

70C3	C3 25 73		JMP	NEXT2
70C6	3D	TYPE3	DCR	A
70C7	C2 EF 70		JNZ	TYPE4
70CA	1A		LDAX	D
70CB	FE 0D		CPI	CR
70CD	C2 B3 70		JNZ	BADOP
70D0	2A A4 73		LHLD	SPSTOR
70D3	F9		SPHL	
70D4	2A 9A 73		LHLD	PCSTOR
70D7	23		INX	H
70D8	E5		PUSH	H
70D9	21 00 00		LXI	H, 0
70DC	39		DAD	SP
70DD	22 A4 73		SHLD	SPSTOR
70E0	31 FF 6F		LXI	SP, STACK
70E3	78		MOV	A, B
70E4	E6 38		ANI	038H
70E6	6F		MOV	L, A
70E7	26 00		MVI	H, 0
70E9	22 9A 73		SHLD	PCSTOR
70EC	C3 6C 72		JMP	SPREAD
70EF	3D	TYPE4	DCR	A
70F0	C2 15 71		JNZ	TYPE5
70F3	1A		LDAX	D
70F4	FE 0D		CPI	CR
70F6	C2 B3 70		JNZ	BADOP
70F9	CD 6E 73		CALL	SHOTEL
70FC	0D 72 65 74 75		DB	CR, "RETURNING TO MONITOR", 0
	72 6E 69 6E 67			
	20 74 6F 20 6D			
	6F 6E 69 74 6F			
	72 00			
7112	CD 40 00		CALL	MONIT

## C-34 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

7115	3D	TYPE5	DCR	A
7116	C2 28 71		JNZ	TYPE6
7119	1A		LDAX	D
711A	FE 0D		CPI	CR
711C	C2 B3 70		JNZ	BADOP
711F	2A A2 73		LHLD	HSTOR
7122	22 9A 73		SHLD	PCSTOR
7125	C3 6C 72		JMP	SPREAD
7128	3D	TYPE6	DCR	A
7129	C2 4F 71		JNZ	TYPE7
712C	1A		LDAX	D
712D	FE 0D		CPI	CR
712F	C2 B3 70		JNZ	BADOP
7132	0E 01		MVI	C, 1
7134	CD 64 73		CALL	UPDATE
7137	CD 59 73		CALL	CTEST
713A	2A A4 73		LHLD	SPSTOR
713D	F9		SPHL	
713E	E1		POP	H
713F	22 9A 73		SHLD	PCSTOR
7142	21 00 00		LXI	H, 0
7145	39		DAD	SP
7146	22 A4 73		SHLD	SPSTOR
7149	31 FF 6F		LXI	SP, STACK
714C	C3 6C 72		JMP	SPREAD
714F	F5	TYPE7	PUSH	PSW
7150	01 00 00		LXI	B, 0
7153	21 00 00		LXI	H, 0
7156	1A		LDAX	D
7157	FE 0D		CPI	CR
7159	C2 74 71		JNZ	HLOOP
715C	CD 6E 73		CALL	SHOTEL
715F	0D 6D 69 73 73		DB	CR, "MISSING ARGUMENT", 0

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-35

```

        69 6E 67 20 61
        72 67 75 6D 65
        6E 74 00
7171  C3 25 73          JMP          NEXT2
7174  1A              HLOOP        LDAX          D
7175  13              INX          D
7176  FE 0D          CPI          CR
7178  CA BA 71        JZ          NOMORE
717B  D6 30          SUI          30H
717D  DA 9A 71        JC          ERROR
7180  FE 0A          CPI          0AH
7182  DA 91 71        JC          HEXOK
7185  D6 07          SUI          007H
7187  FE 0A          CPI          00AH
7189  DA 9A 71        JC          ERROR
718C  FE 10          CPI          010H
718E  D2 9A 71        JNC         ERROR
7191  29              HEXOK        DAD          H
7192  29              DAD          H
7193  29              DAD          H
7194  29              DAD          H
7195  4F              MOV          C,A
7196  09              DAD          B
7197  C3 74 71        JMP          HLOOP
719A  CD 6E 73        ERROR        CALL         SHOTEL
719D  0D 62 61 64 20 DB          CR,"BAD CHARACTER IN OPERAND",0
        63 68 61 72 61
        63 74 65 72 20
        69 6E 20 6F 70
        65 72 61 6E 64
        00
71B7  C3 25 73          JMP          NEXT2
71BA  22 46 72        NOMORE       SHLD         ARGMNT
71BD  F1              POP          PSW

71BE  3D              DCR          A
71BF  C2 E4 71        JNZ          TYPE8

```

# C-36 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

71C2	0E 02		MVI	C, 2
71C4	7C		MOV	A, H
71C5	FE 00		CPI	0
71C7	CA 2C 72		JZ	DOINSTR
71CA	CD 6E 73		CALL	SHOTEL
71CD	0D 41 72 67 75		DB	CR, "ARGUMENT TOO LARGE", 0
	6D 65 6E 74 20			
	74 6F 6F 20 6C			
	61 72 67 65 00			
71E1	C3 25 73		JMP	NEXT2
71E4	3D	TYPE8	DCR	A
71E5	C2 FE 71		JNZ	TYPE9
71E8	EB		XCHG	
71E9	0E 03		MVI	C, 3
71EB	CD 64 73		CALL	UPDATE
71EE	21 F7 71		LXI	H, JBACK
71F1	22 5F 73		SHLD	ADDRES
71F4	C3 59 73		JMP	CTEST
71F7	EB	JBACK	XCHG	
71F8	22 9A 73		SHLD	PCSTOR
71FB	C3 6C 72		JMP	SPREAD
71FE	3D	TYPE9	DCR	A
71FF	C2 2A 72		JNZ	TYPEA
7202	EB		XCHG	
7203	0E 03		MVI	C, 3
7205	CD 64 73		CALL	UPDATE
7208	21 11 72		LXI	H, CBACK
720B	22 5F 73		SHLD	ADDRES
720E	C3 59 73		JMP	CTEST
7211	2A A4 73	CBACK	LHLD	SPSTOR
7214	F9		SPHL	
7215	2A 9A 73		LHLD	PCSTOR
7218	E5		PUSH	H

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-37

7219	21 00 00		LXI	H, 0
721C	39		DAD	SP
721D	22 A4 73		SHLD	SPSTOR
7220	31 FF 6F		LXI	SP, STACK
7223	EB		XCHG	
7224	22 9A 73		SHLD	PCSTOR
7227	C3 6C 72		JMP	SPREAD
722A	0E 03	TYPEA	MVI	C, 3
722C	CD 64 73	DOINSTR	CALL	UPDATE
722F	2A 9C 73		LHLD	PSWSTOR
7232	E5		PUSH	H
7233	F1		POP	PSW
7234	2A 9E 73		LHLD	BSTOR
7237	E5		PUSH	H
7238	C1		POP	B
7239	2A A0 73		LHLD	DSTOR
723C	E5		PUSH	H
723D	D1		POP	D
723E	2A A4 73		LHLD	SPSTOR
7241	F9		SPHL	
7242	2A A2 73		LHLD	HSTOR
7245		OPCODE	DS	1
7246		ARGMNT	DS	2
7248	32 9C 73		STA	PSWSTOR
724B	1F		RAR	
724C	22 A2 73		SHLD	HSTOR
724F	21 00 00		LXI	H, 0
7252	39		DAD	SP
7253	22 A4 73		SHLD	SPSTOR
7256	31 FF 6F		LXI	SP, STACK
7259	17		RAL	
725A	3A 9C 73		LDA	PSWSTOR
725D	F5		PUSH	PSW
725E	E1		POP	H
725F	22 9C 73		SHLD	PSWSTOR

# C-38 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

7262	C5		PUSH	B
7263	E1		POP	H
7264	22 9E 73		SHLD	BSTOR
7267	D5		PUSH	D
7268	E1		POP	H
7269	22 A0 73		SHLD	DSTOR
726C	CD 6E 73	SPREAD	CALL	SHOTEL
726F	0D 50 43 20 20		DB	CR,"PC SZAPC Accumulator A"
	20 53 5A 41 50			
	43 20 41 63 63			
	75 6D 75 6C 61			
	74 6F 72 20 41			
	20 20 42 20 43		DB	" B C D E H L M SP "
	20 20 44 20 45			
	20 20 48 20 4C			
	20 20 4D 20 20			
	53 50 20 20 20			
	53 74 61 63 6B		DB	"Stack",CR,0
	0D 00			
72A8	2A 9A 73		LHLD	PCSTOR
72AB	EB		XCHG	
72AC	CD 7D 73		CALL	HWORD
72AF	CD 54 73		CALL	BLNK1
72B2	2A 9C 73		LHLD	PSWSTOR
72B5	7D		MOV	A,L
72B6	CD 3B 73		CALL	AFLAG
72B9	CD 3B 73		CALL	AFLAG
72BC	07		RLC	
72BD	CD 3B 73		CALL	AFLAG
72C0	07		RLC	
72C1	CD 3B 73		CALL	AFLAG
72C4	07		RLC	
72C5	CD 3B 73		CALL	AFLAG
72C8	CD 54 73		CALL	BLNK1
72CB	7C		MOV	A,H
72CC	F5		PUSH PSW	

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-39

72CD	CD 3B 73	BITWISE	CALL	AFLAG
72D0	CD 3B 73		CALL	AFLAG
72D3	CD 3B 73		CALL	AFLAG
72D6	CD 3B 73		CALL	AFLAG
72D9	F5		PUSH	PSW
72DA	CD 54 73		CALL	BLNK1
72DD	F1		POP	PSW
72DE	CD 3B 73		CALL	AFLAG
72E1	CD 3B 73		CALL	AFLAG
72E4	CD 3B 73		CALL	AFLAG
72E7	CD 3B 73		CALL	AFLAG
72EA	CD 54 73		CALL	BLNK1
72ED	F1		POP	PSW
72EE	CD 54 73		CALL	BLNK1
72F1	CD 54 73		CALL	BLNK1
72F4	5C		MOV	E, H
72F5	CD 81 73		CALL	HBYTE
72F8	CD 54 73		CALL	BLNK1
72FB	2A 9E 73		LHLD	BSTOR
72FE	CD 49 73		CALL	SHOREG
7301	2A A0 73		LHLD	DSTOR
7304	CD 49 73		CALL	SHOREG
7307	2A A2 73		LHLD	HSTOR
730A	CD 49 73		CALL	SHOREG
730D	5E		MOV	E, M
730E	CD 81 73		CALL	HBYTE
7311	CD 54 73		CALL	BLNK1
7314	2A A4 73		LHLD	SPSTOR
7317	EB		XCHG	
7318	CD 7D 73		CALL	HWORD
731B	CD 54 73		CALL	BLNK1
731E	EB		XCHG	
731F	5E		MOV	E, M
7320	23		INX	H
7321	56		MOV	D, M
7322	CD 7D 73		CALL	HWORD
7325	CD 6E 73	NEXT2	CALL	SHOTEL



# C-40 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

7328	0D 49 6E 73 74		DB	CR, "INSTRUCTION? ", 0
	72 75 63 74 69			
	6F 6E 3F 20 20			
	00			
7338	C3 2B 70		JMP	NEXT1
733B	07	AFLAG	RLC	
733C	6F		MOV	L, A
733D	3E 31		MVI	A, "1"
733F	DA 44 73		JC	FLAGON
7342	3E 30		MVI	A, "0"
7344	CD 46 00	FLAGON	CALL	PUTC
7347	7D		MOV	A, L
7348	C9		RET	
7349	5C	SHOREG	MOV	E, H
734A	CD 81 73		CALL	HBYTE
734D	5D		MOV	E, L
734E	CD 81 73		CALL	HBYTE
7351	C3 54 73		JMP	BLNK1
7354	3E 20	BLNK1	MVI	A, " "
7356	C3 96 73		JMP	OUTPUT
7359	2A 9C 73	CTEST	LHLD	PSWSTOR
735C	E5		PUSH	H
735D	F1		POP	PSW
735E		CONDTN	DS	1
735F		ADDRES	DS	2
7361	C3 6C 72		JMP	SPREAD
7364	2A 9A 73	UPDATE	LHLD	PCSTOR
7367	06 00		MVI	B, 0
7369	09		DAD	B
736A	22 9A 73		SHLD	PCSTOR
736D	C9		RET	

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-41

736E	E3	SHOTEL	XTHL	
736F	7E	DISPL	MOV	A, M
7370	23		INX	H
7371	B7		ORA	A
7372	CA 7B 73		JZ	EXIT
7375	CD 46 00		CALL	PUTC
7378	C3 6F 73		JMP	DISPL
737B	E3	EXIT	XTHL	
737C	C9		RET	
737D	7A	HWORD	MOV	A, D
737E	CD 82 73		CALL	DOHEX
7381	7B	HBYTE	MOV	A, E
7382	F5	DOHEX	PUSH	PSW
7383	0F		RRC	
7384	0F		RRC	
7385	0F		RRC	
7386	0F		RRC	
7387	CD 8B 73		CALL	HEXOUT
738A	F1		POP	PSW
738B	E6 0F	HEXOUT	ANI	00FH
738D	C6 30		ADI	030H
738F	FE 3A		CPI	": "
7391	DA 96 73		JC	OUTPUT
7394	C6 07		ADI	007H
7396	CD 46 00	OUTPUT	CALL	PUTC
7399	C9		RET	
739A				
739A		PCSTOR	DS	2
739C		PSWSTOR	DS	2
739E		BSTOR	DS	2
73A0		DSTOR	DS	2
73A2		HSTOR	DS	2
73A4		SPSTOR	DS	2
73A6		BUFFR	DS	10H
73B6	4E 4F 50 02	TABLE	DB	"NOP", 2

## C-42 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

73BA	4C 58 49 20 42 2C 0A	DB	"LXI B",10
73C1	53 54 41 58 20 42 02	DB	"STAX B",2
73C8	49 4E 58 20 42 02	DB	"INX B",2
73CE	49 4E 52 20 42 02	DB	"INR B",2
73D4	44 43 52 20 42 02	DB	"DCR B",2
73DA	4D 56 49 20 42 2C 07	DB	"MVI B",7
73E1	52 4C 43 02	DB	"RLC",2
73E5	3F 3F 3F 3F 01	DB	"????",1
73EA	44 41 44 20 42 02	DB	"DAD B",2
73F0	4C 44 41 58 20 42 02	DB	"LDAX B",2
73F7	44 43 58 20 42 02	DB	"DCX B",2
73FD	49 4E 52 20 43 02	DB	"INR C",2
7403	44 43 52 20 43 02	DB	"DCR C",2
7409	4D 56 49 20 43 2C 07	DB	"MVI C",7
7410	52 52 43 02	DB	"RRC",2
7414	3F 3F 3F 3F 01	DB	"????",1
7419	4C 58 49 20 44 2C 0A	DB	"LXI D",10
7420	53 54 41 58 20 44 02	DB	"STAX D",2
7427	49 4E 58 20 44 02	DB	"INX D",2
742D	49 4E 52 20 44 02	DB	"INR D",2
7433	44 43 52 20 44	DB	"DCR D",2

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-43

	02		
7439	4D 56 49 20 44	DB	"MVI D", 7
	2C 07		
7440	52 41 4C 02	DB	"RAL", 2
7444	3F 3F 3F 3F 01	DB	"????", 1
7449	44 41 44 20 44	DB	"DAD D", 2
	02		
744F	4C 44 41 58 20	DB	"LDAX D", 2
	44 02		
7456	44 43 58 20 44	DB	"DCX D", 2
	02		
745C	49 4E 52 20 45	DB	"INR E", 2
	02		
7462	44 43 52 20 45	DB	"DCR E", 2
	02		
7468	4D 56 49 20 45	DB	"MVI E", 7
	2C 07		
746F	52 41 52 02	DB	"RAR", 2
7473	52 49 4D 02	DB	"RIM", 2
7477	4C 58 49 20 48	DB	"LXI H", 10
	2C 0A		
747E	53 48 4C 44 20	DB	"SHLD ", 10
	0A		
7484	49 4E 58 20 48	DB	"INX H", 2
	02		
748A	49 4E 52 20 48	DB	"INR H", 2
	02		
7490	44 43 52 20 48	DB	"DCR H", 2
	02		
7496	4D 56 49 20 48	DB	"MVI H", 7
	2C 07		
749D	44 41 41 02	DB	"DAA", 2
74A1	3F 3F 3F 3F 01	DB	"????", 1
74A6	44 41 44 20 48	DB	"DAD H", 2
	02		
74AC	4C 48 4C 44 20	DB	"LHLD ", 2
	02		

## C-44 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

74B2	44 43 58 20 48 02	DB	"DCX H",2
74B8	49 4E 52 20 4C 02	DB	"INR L",2
74BE	44 43 52 20 4C 02	DB	"DCR L",2
74C4	4D 56 49 20 4C 2C 07	DB	"MVI L",7
74CB	43 4D 41 02	DB	"CMA",2
74CF	53 49 4D 02	DB	"SIM",2
74D3	4C 58 49 20 53 50 2C 0A	DB	"LXI SP",10
74DB	53 54 41 20 0A	DB	"STA ",10
74E0	49 4E 58 20 53 50 02	DB	"INX SP",2
74E7	49 4E 52 20 4D 02	DB	"INR M",2
74ED	44 43 52 20 4D 02	DB	"DCR M",2
74F3	4D 56 49 20 4D 2C 07	DB	"MVI M",7
74FA	53 54 43 02	DB	"STC",2
74FE	3F 3F 3F 3F 01	DB	"????",1
7503	44 41 44 20 53 50 02	DB	"DAD SP",2
750A	4C 44 41 20 0A	DB	"LDA ",10
750F	44 43 58 20 53 50 02	DB	"DCX SP",2
7516	49 4E 52 20 41 02	DB	"INR A",2
751C	44 43 52 20 41 02	DB	"DCR A",2
7522	4D 56 49 20 41 2C 07	DB	"MVI A",7
7529	43 4D 43 02	DB	"CMC",2
752D	4D 4F 56 20 42 2C 42 02	DB	"MOV B,B",2

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-45

7535	4D 4F 56 20 42	DB	"MOV B,C",2
	2C 43 02		
753D	4D 4F 56 20 42	DB	"MOV B,D",2
	2C 44 02		
7545	4D 4F 56 20 42	DB	"MOV B,E",2
	2C 45 02		
754D	4D 4F 56 20 42	DB	"MOV B,H",2
	2C 48 02		
7555	4D 4F 56 20 42	DB	"MOV B,L",2
	2C 4C 02		
755D	4D 4F 56 20 42	DB	"MOV B,M",2
	2C 4D 02		
7565	4D 4F 56 20 42	DB	"MOV B,A",2
	2C 41 02		
756D	4D 4F 56 20 43	DB	"MOV C,B",2
	2C 42 02		
7575	4D 4F 56 20 43	DB	"MOV C,C",2
	2C 43 02		
757D	4D 4F 56 20 43	DB	"MOV C,D",2
	2C 44 02		
7585	4D 4F 56 20 43	DB	"MOV C,E",2
	2C 45 02		
758D	4D 4F 56 20 43	DB	"MOV C,H",2
	2C 48 02		
7595	4D 4F 56 20 43	DB	"MOV C,L",2
	2C 4C 02		
759D	4D 4F 56 20 43	DB	"MOV C,M",2
	2C 4D 02		
75A5	4D 4F 56 20 43	DB	"MOV C,A",2
	2C 41 02		
75AD	4D 4F 56 20 44	DB	"MOV D,B",2
	2C 42 02		
75B5	4D 4F 56 20 44	DB	"MOV D,C",2
	2C 43 02		
75BD	4D 4F 56 20 44	DB	"MOV D,D",2
	2C 44 02		
75C5	4D 4F 56 20 44	DB	"MOV D,E",2

## C-46 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

2C 45 02			
75CD	4D 4F 56 20 44	DB	"MOV D,H",2
2C 48 02			
75D5	4D 4F 56 20 44	DB	"MOV D,L",2
2C 4C 02			
75DD	4D 4F 56 20 44	DB	"MOV D,M",2
2C 4D 02			
75E5	4D 4F 56 20 44	DB	"MOV D,A",2
2C 41 02			
75ED	4D 4F 56 20 45	DB	"MOV E,B",2
2C 42 02			
75F5	4D 4F 56 20 45	DB	"MOV E,C",2
2C 43 02			
75FD	4D 4F 56 20 45	DB	"MOV E,D",2
2C 44 02			
7605	4D 4F 56 20 45	DB	"MOV E,E",2
2C 45 02			
760D	4D 4F 56 20 45	DB	"MOV E,H",2
2C 48 02			
7615	4D 4F 56 20 45	DB	"MOV E,L",2
2C 4C 02			
761D	4D 4F 56 20 45	DB	"MOV E,M",2
2C 4D 02			
7625	4D 4F 56 20 45	DB	"MOV E,A",2
2C 41 02			
762D	4D 4F 56 20 48	DB	"MOV H,B",2
2C 42 02			
7635	4D 4F 56 20 48	DB	"MOV H,C",2
2C 43 02			
763D	4D 4F 56 20 48	DB	"MOV H,D",2
2C 44 02			
7645	4D 4F 56 20 48	DB	"MOV H,E",2
2C 45 02			
764D	4D 4F 56 20 48	DB	"MOV H,H",2
2C 48 02			
7655	4D 4F 56 20 48	DB	"MOV H,L",2
2C 4C 02			

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-47

765D	4D 4F 56 20 48 2C 4D 02	DB	"MOV H,M",2
7665	4D 4F 56 20 48 2C 41 02	DB	"MOV H,A",2
766D	4D 4F 56 20 4C 2C 42 02	DB	"MOV L,B",2
7675	4D 4F 56 20 4C 2C 43 02	DB	"MOV L,C",2
767D	4D 4F 56 20 4C 2C 44 02	DB	"MOV L,D",2
7685	4D 4F 56 20 4C 2C 45 02	DB	"MOV L,E",2
768D	4D 4F 56 20 4C 2C 48 02	DB	"MOV L,H",2
7695	4D 4F 56 20 4C 2C 4C 02	DB	"MOV L,L",2
769D	4D 4F 56 20 4C 2C 4D 02	DB	"MOV L,M",2
76A5	4D 4F 56 20 4C 2C 41 02	DB	"MOV L,A",2
76AD	4D 4F 56 20 4D 2C 42 02	DB	"MOV M,B",2
76B5	4D 4F 56 20 4D 2C 43 02	DB	"MOV M,C",2
76BD	4D 4F 56 20 4D 2C 44 02	DB	"MOV M,D",2
76C5	4D 4F 56 20 4D 2C 45 02	DB	"MOV M,E",2
76CD	4D 4F 56 20 4D 2C 48 02	DB	"MOV M,H",2
76D5	4D 4F 56 20 4D 2C 4C 02	DB	"MOV M,L",2
76DD	48 4C 54 04	DB	"HLT",4
76E1	4D 4F 56 20 4D 2C 41 02	DB	"MOV M,A",2
76E9	4D 4F 56 20 41 2C 42 02	DB	"MOV A,B",2



## C-48 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

76F1	4D 4F 56 20 41	DB	"MOV A,C",2
	2C 43 02		
76F9	4D 4F 56 20 41	DB	"MOV A,D",2
	2C 44 02		
7701	4D 4F 56 20 41	DB	"MOV A,E",2
	2C 45 02		
7709	4D 4F 56 20 41	DB	"MOV A,H",2
	2C 48 02		
7711	4D 4F 56 20 41	DB	"MOV A,L",2
	2C 4C 02		
7719	4D 4F 56 20 41	DB	"MOV A,M",2
	2C 4D 02		
7721	4D 4F 56 20 41	DB	"MOV A,A",2
	2C 41 02		
7729	41 44 44 20 42	DB	"ADD B",2
	02		
772F	41 44 44 20 43	DB	"ADD C",2
	02		
7735	41 44 44 20 44	DB	"ADD D",2
	02		
773B	41 44 44 20 45	DB	"ADD E",2
	02		
7741	41 44 44 20 48	DB	"ADD H",2
	02		
7747	41 44 44 20 4C	DB	"ADD L",2
	02		
774D	41 44 44 20 4D	DB	"ADD M",2
	02		
7753	41 44 44 20 41	DB	"ADD A",2
	02		
7759	41 44 43 20 42	DB	"ADC B",2
	02		
775F	41 44 43 20 43	DB	"ADC C",2
	02		
7765	41 44 43 20 44	DB	"ADC D",2
	02		

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-49

776B	41 44 43 20 45 02	DB	"ADC E",2
7771	41 44 43 20 48 02	DB	"ADC H",2
7777	41 44 43 20 4C 02	DB	"ADC L",2
777D	41 44 43 20 4D 02	DB	"ADC M",2
7783	41 44 43 20 41 02	DB	"ADC A",2
7789	53 55 42 20 42 02	DB	"SUB B",2
778F	53 55 42 20 43 02	DB	"SUB C",2
7795	53 55 42 20 44 02	DB	"SUB D",2
779B	53 55 42 20 45 02	DB	"SUB E",2
77A1	53 55 42 20 48 02	DB	"SUB H",2
77A7	53 55 42 20 4C 02	DB	"SUB L",2
77AD	53 55 42 20 4D 02	DB	"SUB M",2
77B3	53 55 42 20 41 02	DB	"SUB A",2
77B9	53 42 42 20 42 02	DB	"SBB B",2
77BF	53 42 42 20 43 02	DB	"SBB C",2
77C5	53 42 42 20 44 02	DB	"SBB D",2
77CB	53 42 42 20 45 02	DB	"SBB E",2
77D1	53 42 42 20 48 02	DB	"SBB H",2

## C-50 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

77D7	53 42 42 20 4C	DB	"SBB L",2
	02		
77DD	53 42 42 20 4D	DB	"SBB M",2
	02		
77E3	53 42 42 20 41	DB	"SBB A",2
	02		
77E9	41 4E 41 20 42	DB	"ANA B",2
	02		
77EF	41 4E 41 20 43	DB	"ANA C",2
	02		
77F5	41 4E 41 20 44	DB	"ANA D",2
	02		
77FB	41 4E 41 20 45	DB	"ANA E",2
	02		
7801	41 4E 41 20 48	DB	"ANA H",2
	02		
7807	41 4E 41 20 4C	DB	"ANA L",2
	02		
780D	41 4E 41 20 4D	DB	"ANA M",2
	02		
7813	41 4E 41 20 41	DB	"ANA A",2
	02		
7819	58 52 41 20 42	DB	"XRA B",2
	02		
781F	58 52 41 20 43	DB	"XRA C",2
	02		
7825	58 52 41 20 44	DB	"XRA D",2
	02		
782B	58 52 41 20 45	DB	"XRA E",2
	02		
7831	58 52 41 20 48	DB	"XRA H",2
	02		
7837	58 52 41 20 4C	DB	"XRA L",2
	02		
783D	58 52 41 20 4D	DB	"XRA M",2
	02		

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-51

7843	58 52 41 20 41	DB	"XRA A",2
	02		
7849	4F 52 41 20 42	DB	"ORA B",2
	02		
784F	4F 52 41 20 43	DB	"ORA C",2
	02		
7855	4F 52 41 20 44	DB	"ORA D",2
	02		
785B	4F 52 41 20 45	DB	"ORA E",2
	02		
7861	4F 52 41 20 48	DB	"ORA H",2
	02		
7867	4F 52 41 20 4C	DB	"ORA L",2
	02		
786D	4F 52 41 20 4D	DB	"ORA M",2
	02		
7873	4F 52 41 20 41	DB	"ORA A",2
	02		
7879	43 4D 50 20 42	DB	"CMP B",2
	02		
787F	43 4D 50 20 43	DB	"CMP C",2
	02		
7885	43 4D 50 20 44	DB	"CMP D",2
	02		
788B	43 4D 50 20 45	DB	"CMP E",2
	02		
7891	43 4D 50 20 48	DB	"CMP H",2
	02		
7897	43 4D 50 20 4C	DB	"CMP L",2
	02		
789D	43 4D 50 20 4D	DB	"CMP M",2
	02		
78A3	43 4D 50 20 41	DB	"CMP A",2
	02		
78A9	52 4E 5A 06	DB	"RNZ",6
78AD	50 4F 50 20 42	DB	"POP B",2
	02		

# C-52 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

78B3	4A 4E 5A 20 08	DB	"JNZ ",8
78B8	4A 4D 50 20 08	DB	"JMP ",8
78BD	43 4E 5A 20 09	DB	"CNZ ",9
78C2	50 55 53 48 20	DB	"PUSH B",2
	42 02		
78C9	41 44 49 20 07	DB	"ADI ",7
78CE	52 53 54 20 30	DB	"RST 0",3
	03		
78D4	52 5A 06	DB	"RZ",6
78D7	52 45 54 06	DB	"RET",6
78DB	4A 5A 20 08	DB	"JZ ",8
78DF	3F 3F 3F 3F 01	DB	"????",1
78E4	43 5A 20 09	DB	"CZ ",9
78E8	43 41 4C 4C 20	DB	"CALL ",9
	09		
78EE	41 43 49 20 07	DB	"ACI ",7
78F3	52 53 54 20 31	DB	"RST 1",3
	03		
78F9	52 4E 43 06	DB	"RNC",6
78FD	50 4F 50 20 44	DB	"POP D",2
	02		
7903	4A 4E 43 20 08	DB	"JNC ",8
7908	4F 55 54 20 07	DB	"OUT ",7
790D	43 4E 43 20 09	DB	"CNC ",9
7912	50 55 53 48 20	DB	"PUSH D",2
	44 02		
7919	53 55 49 20 07	DB	"SUI ",7
791E	52 53 54 20 32	DB	"RST 2",3
	03		
7924	52 43 06	DB	"RC",6
7927	3F 3F 3F 3F 01	DB	"????",1
792C	4A 43 20 08	DB	"JC ",8
7930	49 4E 20 07	DB	"IN ",7
7934	43 43 20 09	DB	"CC ",9
7938	3F 3F 3F 3F 01	DB	"????",1
793D	53 42 49 20 07	DB	"SBI ",7

# PROGRAM CARTRIDGE SOURCE CODE LISTINGS C-53

7942	52 53 54 20 33	DB	"RST 3",3
	03		
7948	52 50 4F 06	DB	"RPO",6
794C	50 4F 50 20 48	DB	"POP H",2
	02		
7952	4A 50 4F 20 08	DB	"JPO ",8
7957	58 54 48 4C 02	DB	"XTHL",2
795C	43 50 4F 20 09	DB	"CPO ",9
7961	50 55 53 48 20	DB	"PUSH H",2
	48 02		
7968	41 4E 49 20 07	DB	"ANI ",7
796D	52 53 54 20 34	DB	"RST 4",3
	03		
7973	52 50 45 06	DB	"RPE",6
7977	50 43 48 4C 05	DB	"PCHL",5
797C	4A 50 45 20 08	DB	"JPE ",8
7981	58 43 48 47 02	DB	"XCHG",2
7986	43 50 45 20 09	DB	"CPE ",9
798B	3F 3F 3F 3F 01	DB	"????",1
7990	58 52 49 20 07	DB	"XRI ",7
7995	52 53 54 20 35	DB	"RST 5",3
	03		
799B	52 50 06	DB	"RP",6
799E	50 4F 50 20 50	DB	"POP PSW",2
	53 57 02		
79A6	4A 50 20 08	DB	"JP ",8
79AA	44 49 02	DB	"DI",2
79AD	43 50 20 09	DB	"CP ",9
79B1	50 55 53 48 20	DB	"PUSH PSW",2
	50 53 57 02		
79BA	4F 52 49 20 07	DB	"ORI ",7
79BF	52 53 54 20 36	DB	"RST 6",3
	03		
79C5	52 4D 06	DB	"RM",6
79C8	53 50 48 4C 02	DB	"SPHL",2
79CD	4A 4D 20 08	DB	"JM ",8

## C-54 PROGRAM CARTRIDGE SOURCE CODE LISTINGS

---

79D1	45 49 02	DB	"EI",2
79D4	43 4D 20 09	DB	"CM ",9
79D8	3F 3F 3F 3F 01	DB	"????",1
79DD	43 50 49 20 07	DB	"CPI ",7
79E2	52 53 54 20 37	DB	"RST 7",3
	03		
79E8			

# INDEX



**A**

A flag	2-33
Accumulator	1-8
Add	2-13
Address bus	1-9
Address register	1-16
ALU	1-14, 2-6
AND	2-24
ASCII	1-9
Assemblers	4-6
Assembler directives	5-18
Assembly language	3-7
Auxiliary carry	2-9

**B**

Base port address	6-6
Binary coded decimal	2-31
Bits	1-8, 2-7
Borrow	2-22
Branches	4-24
Bus	1-5
Byte	1-8

**C**

Carry	2-9, 2-14, 2-21, 2-34
Calls	5-10
Circle	4-13
CMA	2-31, 2-35
Coding	4-18
Compilers	4-6
Complement the accumulator	2-35
Conditional calls	5-11
Conditional jumps	4-22
Conditional returns	5-11
Condition codes	4-23
Connection	4-13
Controller-sequencer	1-16
CSEG	5-20

**D**

DAA	2-31, 2-33
DAD SP	5-7
Data bus	1-9
Decision	4-11
Diamond	4-11
Direct addressing	3-10
Direct loads and stores	3-10
DSEG	5-20

**E**

Exchange	3-13
Exclusive OR	2-28
Execute	1-20, 1-32

**F**

Fetch	1-20, 1-27
FIFO	5-5
Flags	4-23
Flowchart construction	4-15
Flow charts	4-9

**G**

Greater than	4-12
--------------	------

**H**

Halt (See halt)	
High level languages	4-5
HLT	1-37

**I**

Immediate	2-17
Immediate addressing	3-7
<u>IN</u>	6-8
<u>INTA</u>	6-15
INTR	6-15
Index	2-16
Indirect loads and stores	3-9
Input	1-6, 6-11
Instruction	1-6
Instruction decoder	1-16
Instruction set	2-10
Interpreters	4-6
Interrupts	6-14
I/O	1-6
I/O device	1-6
<u>IO/M</u>	6-5
I/O port	1-6

**J**

Jumps 4-22, 4-24

**L**

LIFO 5-6

Labels 3-7

Languages 4-5

Less than 4-12

Loads 3-9

Loops 4-24

Low level languages 4-5

**M**

Masking 2-25

Memory 1-16

Microcomputer 1-5

Microprocessor 1-5

Minuend 2-21

Mnemonic 1-21

Most significant bit 1-10

Most significant bit 1-10

Move instructions 3-5

MPU 1-5

**N**

Nibble 1-17

NOT 4-12

NOP 6-12

**O**

Object code 4-6

Opcode 1-21

Operand 1-14

Operations 4-14

OR 2-26

ORG 5-20

OUT 6-5

Output 1-6, 6-12

Outside world 1-6

Oval 4-10

**P**

Parity	2-9
Planning	4-8
Pop	5-16
Port	6-5
Processor status word	2-8
Program	1-6, 4-5
Programming model	2-9
Programming languages	4-5
Program counter	1-16
Pseudo operations	5-18
Push	5-16

**Q**

Queue	5-5
-------	-----

**R**

RAM	1-19
$\overline{\text{RD}}$	6-5
Read	1-18
Rectangle	4-14
Recursion	5-14
Recursive	5-14
Register	1-9
Register transfers	3-12
Register array	2-7
Reset	6-18
Restart	6-18
Returns	5-10
RIM	6-10
ROM	1-19
Rotate around left	2-35
Rotate around right	2-35
Rotate left without carry	2-36
Rotate right without carry	2-36
RST	5-12
RST x.5	6-16

**S**

Serial output data	6-12
Serial output enable	6-12
SID	6-10
Sign	2-9
SIM	6-10
SOD	6-10, 6-12
SOE	6-12
Source code	4-6
Stack	5-5
Stack Pointer	5-6
Stores	3-9
Stored program concept	1-6
Subroutines	5-10
Subtract	2-20
Subtrahend	2-21

**T**

Terminal	4-10
Transfers	3-12
TRAP	6-14

**U**

Unconditional jumps	4-21
---------------------	------

**W**

$\overline{\text{WR}}$	6-5
Word	1-8
Write	1-17

**Z**

Zero	2-9
------	-----